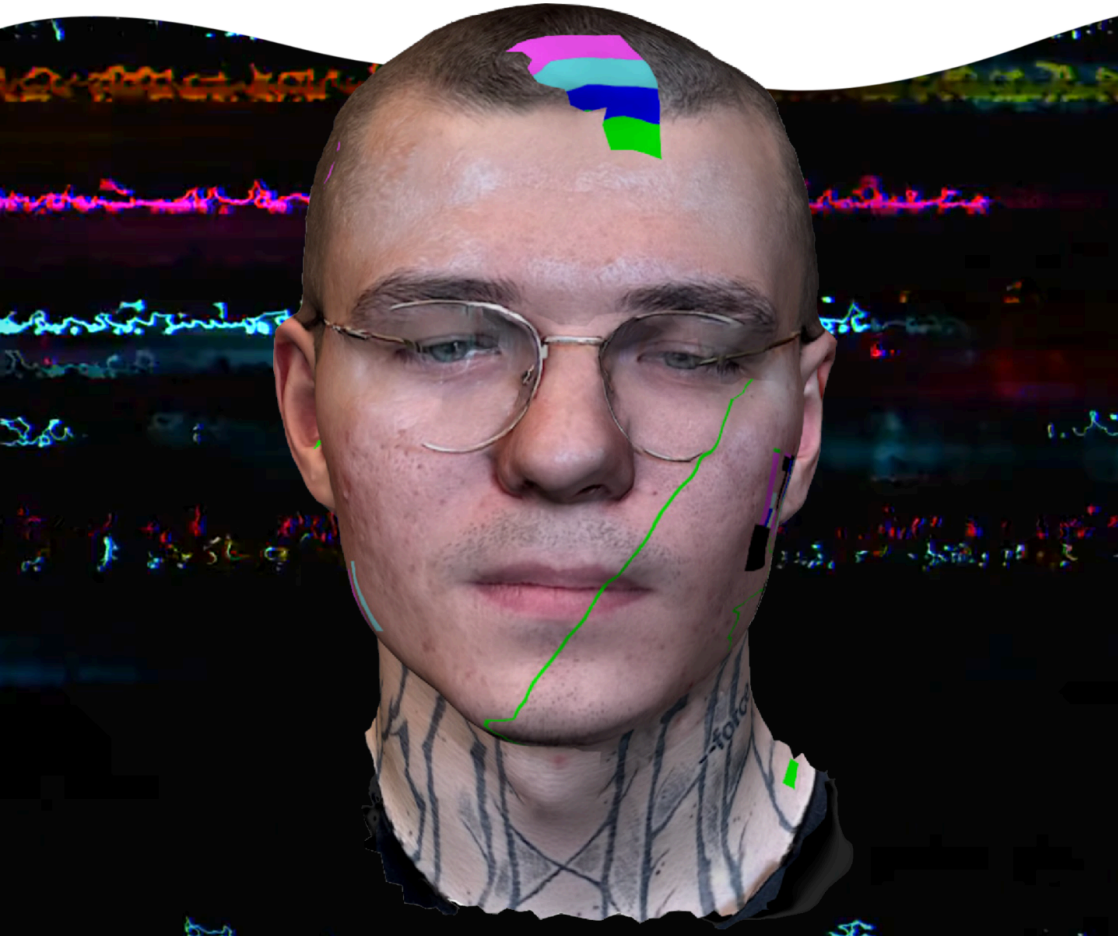
ЯЮниор AGAVA ROMAN SAKUTIN

C# ДЛЯ НАЧИНАЮЩИХ НА ПРАКТИКЕ



Роман Сакутин. 1-ое издание

Том 1. Вендинговый автомат

Введение

Эта книга распространяется бесплатно в электронном виде. Это **первое** издание. Все актуальные обновления и новые издания ищите в моём телеграмм-блоге.

Связь со мной:

Email - romaan27@gmail.com
Телеграмм - @rsakutin

Об Авторе

Меня зовут Роман Сакутин.

Я программирую всю свою сознательную жизнь и к текущему моменту имею 10 лет опыта коммерческой разработки.

Мой родной дивизион - это GameDev. В нём я проработал большую часть времени. Но волей судьбы несколько лет я работал как Backend разработчик на ASPNet, а также сертифицирован как PHP программист.

Я веду блоги на YouTube и в телеграмм. Обязательно подписывайтесь!

В какой-то момент у меня появилась школа ЯЮниор, в которой мы готовим сильных программистов, потому что в весь процесс обучения я вносил весь тот опыт, который добыл сам своими руками за годы практики.

У меня есть студия разработки игр AGAVA, которая занимается аутсорс разработкой и созданием своих небольших проектов. Когда-нибудь мы построим большую команду с хорошим бюджетом и сделаем свой проект мечты.

Посвящаю

Посвящаю моим подписчикам, которые 6 лет назад скинулись на написание этой работы.

Я рад каждому дню, что я провожу с вами и для вас. Спасибо вам за вашу поддержку! Если бы не вы, то всего бы этого не было.

Оглавление

Введение.....3
Об Авторе..... 3
Посвящаю..... 3
Оглавление.....3
Структура книги..... 7
Книга не только про алгоритмы.....12
Разделы и задачи..... 13
Устойчивые промежуточные формы..... 14
Функциональная композиция..... 17
Однородность..... 19
Отличие функции от процедуры.....20
Краткий курс C#.....21
Операторы выражений..... 22
Типизация..... 24
Арифметические операторы..... 30
Конкатенация строк.....31
Логические операторы..... 33
Операторы отношений..... 35
Переменные..... 40
Операторы управления.....43
Структурная парадигма..... 44
Область видимости..... 47
Условные операторы..... 50

If....51
If и логические операторы....52
If-else-else if....54
Операторы циклов....57
while....58
for....59
Вырожденные операторы....61
Массивы....62
Использование массивов....66
Перебор массивов циклами....68
Задачи для практики массивов....69
Тонкости инициализации....70
N-мерные массивы....71
Массивы массивов....73
Методы....74
Спуск на один шаг....78
Синтаксис методов....80
Вызов метода....81
Объявление метода....83
Поток выполнения....86
Модификаторы параметров ref и out....88
Необязательные аргументы....96
Эволюция API....100
Функциональная композиция....101
Классы (ООП?)....107
Класс и объект....107
Члены и создание объектов....110

Поля..... 111
Методы..... 114
Инкапсуляция..... 116
Конструкторы..... 122
Связь объектов (Has-a)..... 125
Инкапсуляция..... 127
Наследование (Is-a)..... 133
Цепочки вызовов конструкторов..... 136
Модификатор доступа protected..... 139
Абстракция..... 140
Смысловая абстракция..... 140
Формальная система типов и абстракция..... 144
Виртуальные методы..... 146
Задача с собеседования..... 154
Интерфейсы и абстрактные методы..... 157
Вендинговый автомат..... 164
Описание проекта..... 164
Массивы и циклы..... 167
Ваша задача..... 167
Массивы и циклы.Возможное решение..... 170
Реализация AddMoney..... 173
Реализация GetChange..... 177
Реализация BuyGood {id} {count}..... 177
Функциональная композиция..... 191
Ваша задача..... 191
Избавиться от дублирующегося кода..... 196
Сокрыть структуру данных и сделать

программу понятней.....212
Объектно ориентированный дизайн.....234
Ваша задача.....235
Возможное решение.....235
Основная логика.....237
Роутинг и команды.....248
Куда делся GoodStorage и OrderDeliver?.....261
Принцип подстановки Барбары Лисков на примере заказов.....262
Стоит ли изменять тип параметра в методе TryProcessOrder.....269
Рефлексия на примере команд.....271
Разрешение зависимостей.....279
Позиционная ассоциация аргументов с параметрами.....292
Наделяем команды описанием через атрибуты 295
Боремся с дубликатами с помощью Roslyn API.. 299
Выдача сдачи. Часть 1.....312
Ваша задача.....312
Подсчет чисел Фибоначчи.....313
Реализация формулы через рекурсию.....314
Динамическое программирование.....316
Оптимальная подструктура.....317
Перекрывающиеся подзадачи.....318
Два пути.....319
Нисходящий – мемоизация.....319

Восходящий – табуляция..... 321
Возможное решение. Алгоритм выдачи сдачи с оптимизацией на C#.....323
Решение на основе деления..... 324
Рекурсивное решение..... 326
Рекурсивное решение – оптимизация на основе мемоизации..... 329
Рекурсивное решение – оптимизация на основе табуляции..... 331
Завершение..... 343

Структура книги

Большая часть времени работы над книгой ушло на это. Написать книгу не сложно тогда, когда представляешь какой она будет. В процессе написания я находил новые формы, которые, на мой взгляд, лучше решают поставленную передо мной и этой книгой задачу.

Книга крутится вокруг практики и задач, которые вы обязаны уметь решать. К сожалению, такие интересные темы, как теория графов, раскрываются слегка сухо и студенты в конце обучения слабо представляют где и как это можно было применить.

Так, например, у меня была странная ситуация. Ко мне в группу по обучению попал человек – Юра. Он талантливый ученик и показывает большие успехи в своём вузе. О графах не только наслышан, но и успел обработать много задач в учебном заведении.

У меня он занимался разработкой игр на Unity. Позже его определил в команду, которая писала игру “Вычислитель”. Суть игры в том, что игроку загадывается число и на экран выводится набор выражений. Часть из них в результате даёт загаданное число. И игроку нужно на скорость

решать выражения в уме и выбирать те, ответ на которые был ему загадан.

Я думаю вы уже понимаете, что одна из главных задач для программиста в этом проекте – это написание генератора выражений? То есть нам нужно получить некий модуль, которому остальные члены команды могли бы передать на вход:

1. Необходимый ответ;
2. Список разрешенных операторов;

Под списком операторов подразумевается набор из +, -, *, /. Модуль генерирует такое выражение, в котором использовались бы все возможные операторы из списка (в рандомном порядке), а результат которого совпадал бы с первым аргументом.

Например:

Вход:

1. 25;
2. +, +, -;

Выход:

1. 35 – 20 + 5 + 5;

Я отдал эту задачу Юре. И, к сожалению, он не смог с ней справиться с первого захода. После чего мы

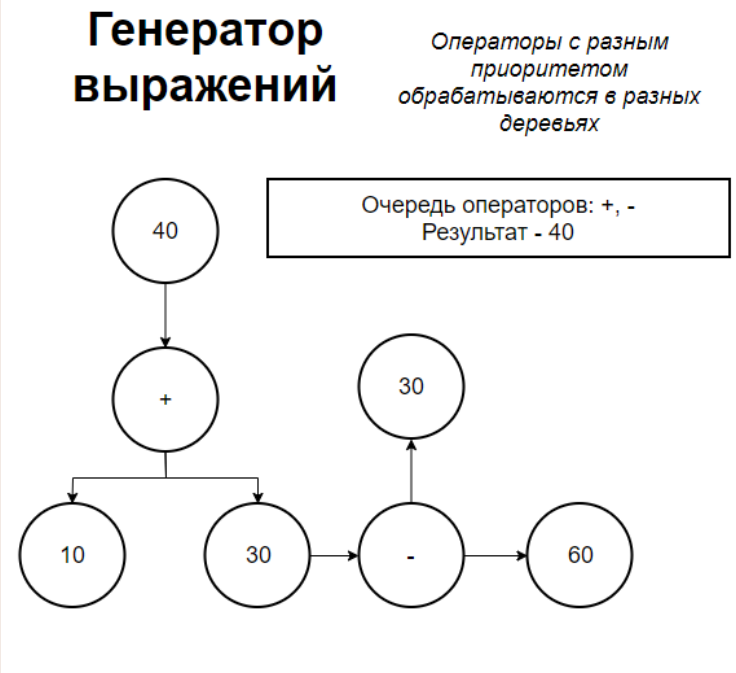
сели с ним и устроили мозговой штурм. И мы, конечно, пришли к решению в поле теории графов.

Это решение у меня было в голове потому, что в начале обучения прочитал книгу по теории компиляции и интерпретации. Там я прочел познавательную главу о редукции и интерпретации выражений. Суть в том, что выражение разбивается на специальное дерево, по которому потом вычисляется.

Например, для выражения:

10 + 60 – 30 = 40;

Мы можем построить такое дерево:



Понимая выражение как граф, мы напишем генератор, смысл которого заключается в следующем: от числа лежит направленное ребро до оператора, результат которого даёт это число; от оператора, так как они все бинарные, должны лежать два направленных ребра до чисел. И так до бесконечности.

Алгоритм работает просто:

1. Ставит корнем дерева число, которое должно получится в итоге;

2. Проводит ребро от этого числа до любого оператора из списка;
3. Генерирует два случайных числа, применение оператора к которым даст нужный результат и проведёт два ребра от оператора до этих чисел;
4. Если есть ещё операторы, то соответственно, ведёт ребра от этих чисел до операторов;

Потом это дерево переводится в читаемый человеком формат. Тут также встает интересующий нас вопрос на счёт приоритета операторов. Он также решаем, но уже не так интересен на данный момент.

Юра, хоть и мог спокойно запрограммировать это решение, и обучен всем необходимым знаниям, чтобы прийти к этому решению, сделать этого не мог.

Тем не менее существует необъятное количество тем, которые могут значительно облегчить жизнь и поднять на новый уровень профессиональных успехов, которые остаются за бортом из-за того, что вы не умеете применять их на практике.

Эта книга структурирована так из-за того, что каждая тема разбирается по несколько раз на различных примерах, открывая вам новые грани возможностей.

Книга состоит из 10 проектов, каждый из которых состоит из шагов, которые проделываются для достижения цели. Подобрав очень интересные задания для вас, называя их “Задачами”, нам предстоит разобрать 100 примеров.

Книга не только про алгоритмы

В предыдущей главе вы могли проследить идею, что книга о задачах, связанных с математикой, но это не так. Книга ориентирована на проектирование и правильное написание кода.

Дело в том, что проблема Юры состояла не только в том, что он не спроецировал знания, приобретенные в вузе, на задачу. А в том, что он написал работающее решение, но с которым никто, кроме него, не мог работать.

Я знал, что это может произойти и сразу, как только мы пришли к тому, как будем решать эту задачу, я прислал ему набор интерфейсов (Те, что в C#), чтобы он мог их реализовать на основе того, что мы придумали.

А сам, опираясь на эти интерфейсы, раздал задачи другим разработчикам и написал свой код. Чтобы, как только Юра закончил, мы могли пройтись

автоматизированными тестами по результатам работ, да и воткнуть в игру и поиграться всей командой. Но, к сожалению, Юре это не удалось.

Он не понимал идей, которые таятся за моей просьбой и, конечно, выкинул интерфейсы и сделал всё на статических методах, с которыми сложно было работать. В результате интеграция его генератора заняла гораздо больше времени, чем хотелось бы. Мне пришлось применить паттерн “Адаптер” и всё встало на свои места. Но кодовая база стала гораздо запутанней.

Поэтому эта книга начинает решение каждого проекта примитивными способами, а потом приводит вас к объектно-ориентированному варианту. Чтобы вы на эволюционном примере понимали таящийся в этом смысл, чтоб в дальнейшем писали хороший и легко поддерживаемый код.

Разделы и задачи

Главный материал книги заключается в практических разделах. Раздел объединяет, сквозной темой, ряд глав.

Каждый раздел – это новая задача, которую нам предстоит решить. Глава разбивает эту задачу на подзадачи и решает их. Иногда, в процессе решения,

мы приходим к узкому месту, которое не можем решить в поле той главы, в которой находимся, именно тогда всё выливается в новую главу, которая вводит новые понятия и с помощью них решает проблему.

Название глав могут повторяться в разных разделах. Это нормально. Воспринимайте название главы как сокращение от полного названия, состоящего из номера тома, раздела и самого названия. Например: “Том 1. Вендинговый автомат. Функциональная композиция”.

Каждая новая глава в каждом разделе постепенно наращивает сложность так, чтобы если вам показалась слишком легкой первая глава раздела, то ближе к концу вы будете находить всё более сложные моменты, которые выводят вас на новую ступеньку мастерства.

Устойчивые промежуточные формы

“Любая работоспособная сложная система является итогом эволюции более простой работоспособной системы... Сложная система, разработанная «с нуля», никогда не работает так, как надо и никакие «заплатки» не заставят ее работать правильно. Проектирование следует начинать с простой работоспособной системы.”

Буч Г.

Все примеры, которые мы будем рассматривать, будут рассказывать о том, как можно естественным образом прийти к тому или иному состоянию системы и что нас приводит к этому. Итак, перед тем как мы приступим к объектно-ориентированному состоянию системы, мы опишем ее поведение максимально простым и быстрым способом.

По мере разработки мы будем встречаться с различными проблемами, которые я хочу решить вместе с вами. Такой способ работы поможет сделать так, чтоб примеры выглядели не натужно и к месту.

Программирование не единственное мое увлечение, но пока что главное. До этого я интересовался построением бизнеса, принципами банковского дела и маркетинга и, самое главное, тем, как создавать продукты и управлять проектами.

Одна из главных мыслей, которые я тогда получил в одной из книг курса MBA, выражается аббревиатурой – MVP. В ней скрыто много интересного, расшифровывается она как “Minimum viable product” или на русском “Минимальный жизнеспособный продукт”.

Я уверен, многие читатели и так знакомы с этой идеей, а если нет, я не хочу становиться в позицию гуру, который вам раскроет какую-либо истину. На самом деле вы и так каждый день живете с этой аббревиатурой. Например, когда вы не хотите делать домашнюю работу в своем месте учёбы, ленитесь помыть посуду или же отказываетесь от какого-нибудь заказа или книги – вам просто это и не нужно, вы и без этого достигаете тех целей, которые сейчас перед вами.

Идея в этом и заключается – сделать что-то минимально жизнеспособное, чтобы проверить, будет ли это работать. Имеет ли это смысл. Действительно ли та форма, которую я собираюсь сделать оптимальна.

Простота будет заключаться не в том, чтобы начать разработку машины с колеса. А в том, чтобы начать с самоката. Самокат может решить вашу задачу, и по мере увеличения требований он превратится в

мотоцикл, и следом в гоночный болид, а может и грузовую гужевую повозку.

Разработка сложной системы состоит в том, чтобы увеличивать сложность постепенно. Отказываясь от не нужного и не самого выгодного.

Функциональная композиция

Мы использовали элементарные арифметические операции, комбинировали их и абстрагировали получившиеся составные операции путем определения составных процедур. Но всего этого еще недостаточно, чтобы сказать, что мы умеем программировать.

Положение, в котором мы находимся, похоже на положение человека, выучившего шахматные правила, но ничего не знающего об основных дебютах, тактики и стратегии.

Подобно шахматисту – новичку, мы пока ничего не знаем об основных схемах использования понятий в нашей области знаний. Нам недостает знаний о том, какие именно ходы следует делать (какие именно процедуры имеет смысл определять), и не хватает опыта предсказания последствий сделанного хода (выполнения процедуры).

Способность предвидеть последствия рассматриваемых действий необходима для того, чтобы стать квалифицированным программистом, — равно как и для любой другой синтетической, творческой деятельности.

Например, квалифицированному фотографу нужно при взгляде на сцену понимать, насколько темным каждый ее участок покажется после печати при разном выборе экспозиции и разных условиях обработки. Только после этого, можно проводить обратные рассуждения и выбирать кадр, освещение, экспозицию и условия обработки так, чтобы получить желаемый результат.

Чтобы стать специалистами, нам надо научиться представлять процессы, генерируемые различными типами процедур. Только развив в себе такую способность, мы сможем научиться надежно строить программы, которые ведут себя так, как нам надо. Структура и интерпретация компьютерных программ.

В ряде разделов я использую главу с названием “функциональная композиция” и мне следует объяснить, что я под этим подразумеваю. Композиция – это построение чего-то сложного из чего-то простого. Стоит понимать, что сложность – это контекстуальное определение. В неких рамках что-то сложное может представать более простым, в результате чего мы можем построить нечто еще более сложное.

Композиция позволяет нам строить сложную иерархическую систему, в которой каждый уровень

максимально понятен и независим от других. И один из способов построения абстракции – это функция или один из частных случаев – метод. С помощью методов мы можем строить слои и разбивать программу на различные кусочки. Чёткое проведение границ позволяет следовать принципу “разделяй и властвуй”.

Когда мы говорим, что некая функция работает с другими функциями, мы говорим о композиции. Так функции низшего порядка позволяют строить более сложные функции и помогают избавиться им от дублирования логики. С каждым шагом мы будем подниматься по уровням, и на самом верхнем будем оперировать понятиями решаемой задачи. С помощью абстракций от деталей мы описываем язык предметной области, на котором и записываем основную логику программы.

Однородность

Когда мы работаем на одном уровне абстракции, нам не следует опускаться с него слишком низко или подниматься слишком высоко. Когда мы говорим об иерархической системе, на одном уровне иерархии должны быть однородные сущности, и они не должны работать со слишком низким уровнем или слишком высоким.

Плохая идея в сущности банковского аккаунта работать с сокетами, хоть это и решает некоторую задачу. Даже если вы будете работать с абстракцией ISocket – ничего не изменится. Даже если вы работаете с абстракцией, то это не значит, что вы работаете со слоем своего уровня.

Отличие функции от процедуры

В языке C# нет понятия процедуры, но есть понятие функции и метода. Функция – это более общее понятие, которое включает в себя метод. Помимо этого, функцией также является анонимный метод и лямбда выражение.

Чаще всего понятие функции и процедуры является взаимозаменяемым. Но при более строгом обсуждении под функциями обычно подразумевается чистая функция (Pure Function), её особенностью является то, что она детерминирована от своих параметров и не несет побочных эффектов.

Процедуре же позволительно работать, как и с глобальным состоянием, так и возвращать результат, зависящий от него. Помимо прочего, она вообще может не иметь результата как такового. Это не означает, что мы не можем создавать композицию процедур. Всё-таки мы освобождены от математического определения этого термина и

можем основывать свое поведение не только на возвращаемом значении процедуры, но и поведении, которое она определяет.

Функция логирования вполне может быть основана на процедуре записи в файл, хотя понятно, что ей ничего не нужно, кроме непосредственного выполнения и записи в файл. При этом тот, кто работает с функцией логирования, не особо заинтересован в тонкостях этого процесса.

В конечном итоге, зачастую функции и процедуры для нас являются равноправным способом построения абстракций и различия нас не особо интересуют, гораздо важней то, что их объединяет.

Краткий курс C#

В этом разделе вы узнаете основной синтаксис, который будет использоваться в дальнейшем. К сожалению, этот раздел не является исчерпывающим и не закрывает задач научить вас программировать с нуля. Читая его, вы уже должны знать многие вещи и иметь хотя бы базовый навык, уметь самостоятельно писать простые программы.

Данный раздел служит краткой справкой, к которой вы можете обращаться в момент чтения других разделов, а также в своей повседневной практике. Также этот раздел имеет общую структуру, схожую со структурой индивидуальных занятий, которые я провожу с людьми, которые занимаются программированием с нуля.

В первую очередь, я хочу рассказать вам о вычислении выражений, далее научить вас строить шаги линейного алгоритма с их помощью. После этого, я покажу вам возможность ветвления и зацикливания собственной логики. Затем, вам нужно расширить свой арсенал с помощью функций, которые позволят вам декомпозировать вашу программу.

Сразу после, мы перейдём к ООП и связанным синтаксическим темам. Всё это время мы будем затрагивать идеологические причины той или иной возможности, которую дает вам язык C#. Сам по себе краткий курс не поставит вам навыков, а они как раз-таки и важны. Навык вы сможете наработать, выполняя задачи уже из основных разделов книги, а также постоянно программируя.

Операторы выражений

Выражение – это основная единица абстракции, с которой вам предстоит работать. Она состоит из операндов и операторов. В выражении 2 + 1 мы имеем два операнда (1 и 2) и один оператор +. Мы можем воспринимать операторы как некие операции, а операнды как некоторые данные, над которыми производится операция.

В случае с 2 + 1 компьютер произведёт сложение и мы получим его результат. Результат выражения – это важная концепция, которая позволяет комбинировать выражения в более сложные. Так 2 + 1 имеет в результате 3.

Мы можем добавить еще один оператор и записать 2 + 1 + 4. Такое выражение будет состоять из нескольких подвыражений. Сначала вычисляется 2 + 1, результатом будет 3. Из-за этого мы получим следующее выражение 3 + 4, результатом которого уже будет итоговое 7.

Выражения имеют много особенностей, которые делают это невероятно интересной темой. Например, вам не нужно уметь отличать аппликативную стратегию вычисления от вызова по значению. Но вам точно нужно понимать, что в C# мы трепетно

относимся к типам, и то,что у нас есть стандартный набор операторов, которыми вы должны уметь пользоваться в рамках примитивных типов.

Операторы выражений делятся на три категории по количеству операндов:

1. Унарные – работают с одним операндом;
2. Бинарные – работают с двумя операндами;
3. Тернарный – работают с тремя операндами;

Типизация

В основании больших и отказоустойчивых программ находится система типов, которая позволяет снизить риск ошибок и выражать наши мысли в коде более точно. В примитивном смысле мы можем говорить о типизации, как о наличие в языке некоего осмысления того, что каждое значение так или иначе имеет некие ограничения и допустимые операции.

В целом, почти все языки, о которых вы знаете, используют одну систему, так называемую “формальную систему типов”. Разнятся в языках только некоторые подходы и детали. Нельзя опускаться в священную войну и пытаться сравнивать это всё в формате “лучше или хуже”. Каждое решение ведёт к своим последствиям при разработке программы.

Если мы рассматриваем некоторые данные, например строку “Привет”, то эти данные связаны с каким-то типом. Для компьютера эта связь не имеет смысла, для него всё представляется нулями и единицами. Эта связь имеет смысл для нас и для некоторых инструментов, которыми мы пользуемся. В случае со строкой “Привет” мы можем оперировать типом “string”, он говорит что перед нами строка состоящая из символов и что символы в формате

Unicode (2 байта на символ). Далее мы можем попробовать совершить такие действия:

“Привет” * 2;

О да, мы описали выражение, в котором пытаемся умножить строку на два. Какой результат у этой операции? На самом деле нельзя сказать. Нет никакой общепринятой аксиомы, как в случае с числами. Технически мы можем описать такую операцию и сказать, что произведение строки и числа даёт новую строку, в которой каждая буква исходной повторяется на основе множителя. А можно придумать другую операцию.

Если такая операция не описана в рамках нашей программы, то она считается ошибочной. О чём нас предупредит наша среда разработки. Каждое выражение также имеет определённый тип, при этом итоговый тип не всегда тот же, что и тип операндов. Главное запомнить, что каждое значение, которое мы рассматриваем, имеет некий тип, с которым мы должны считаться.

В C# есть набор примитивных типов, которые мы можем использовать. Также есть различные пользовательские типы, которые мы можем брать как со стандартной библиотеки, так и описывать самостоятельно. При разговоре про примитивные

типы нельзя опускать такой термин как литерал. Литерал – это прямое представление значения определённого типа. У каждого типа свой формат записи литералов.

Так для целочисленного типа мы записываем просто число. А значение строкового типа описываем в двойных кавычках.

Индентификатор	Литерал	Детали
byte, short, ushort, int, uint, long, ulong	Просто число. Пример: 1, 5, 25, -34, 0. Минус нуля в C# нет;	Целочисленные типы различаются по размеру (8, 16, 32 и 64 бита на число) и доступности отрицательных значений.
bool	true или false;	Логический тип который удобен при работе с логическими операциями. Ограничен двумя значениями но при этом ему нужно 8 бит для их представления.

char	Символ в одинарных кавычках. Пример: ‘a’, ‘b’, ‘1’;	Занимает 16 бит, и представляет символ из таблицы Unicode. Стоит понимать что символ ‘1’ и число 1 – это разные вещи.
string	Некий набор символов (char) который образует строку. Символы все вместе находятся под двойным кавычками. Пример: “Hello”, “12”;	

float, double	Для double просто число с дробной частью через точку. Для float тоже самое, но с припиской f. Пример: 12.0, 12.4, 14.2f;	Необходим для представления нецелых чисел.

Арифметические операторы

Над числовыми типами определены арифметические операции, их операнды числа и результатом является некоторое число. Мы можем комбинировать в выражениях с арифметическими операторами как целочисленные типы так и типы с плавающей запятой, при этом все значения будут переводится в тип с плавающей запятой.

Например в выражении “1 + 1.2” результатом будет “2.2.” При вычисление значение 1 будет рассматриваться под типом double. В некоторых ситуациях этого не происходит, хотя вам может это показаться.

Например, операция “5 / 10” должна дать нам “0.5”. Но вы используете в таком выражении тип int, а он не подразумевает остаток, в результате у нас будет число 0.

Избежать этого можно приписав к числам постфикс, который укажет им более подходящий тип. Например f, который даст тип float или d который даст тип double. “5d / 10d = 0.5”.

Опера тор	Описание	Пример
+	Складывает два числа;	2 + 2 = 4;
–	Вычитает одно число из другого;	4 – 2 = 2;
*	Умножает два числа;	4 * 2 = 8;
/	Делит одно число на другое;	14 / 5 = 2;
%	Возвращает остаток от деления одного числа на другое;	14 / 5 = 4;

Конкатенация строк

Некоторые операторы, хоть и выглядят как арифметические, примененные к не числовым типам имеют другое значение. В C# мы можем перегрузить оператор для определённого типа. Например, для типа string перегружен оператор “+”, что даёт нам соединения двух строк.

Пример:

“Hello ” + “World” = “Hello World”;

С ним стоит быть аккуратней, так как строка + число дает строку. В таком выражении число конвертируется в строку.

Пример:

“Number = ” + 10 = “Number = 10”;

Или такое:

“10” + 10 = “1010”;

Иногда нам хочется в одном выражении вычислить два значения и соединить их со строкой. В таком случае у нас может возникнуть проблема.

Например:

“Result = ” + 10 + 10;

Мы ожидаем строку “Result = 20”, но фактически увидим “Result = 1010”. Почему? Если разбирать по порядку, то сначала выполнится эта часть выражения

“Result = “ + 10;

Мы знаем, что если слева или справа от плюс строка, то противоположная сторона тоже становится строкой.

В итоге мы получим выражение:

“Result = ” + “10” = “Result 10”;

Потом переходим к следующей части:

“Result = 10” + 10;

Тут это всё повторяется и в результате мы видим такой интересный результат. Починить это всё можно с помощью круглых скобок, которые задают приоритет.

“Result = ” + (10 + 10);

В таком случае мы сначала сложим две десятки, а потом присоединим их к строке.

Логические операторы

Операторы этой группы нужны для работы с логическими типами и различными более сложными блочными операторами. Они выражают некоторые операции булевой алгебры, которые необходимы для выражения условий внутри программы. Операнды этих операторов должны иметь тип bool, результатом также будет bool.

Оператор	Описание	Пример
&&	Логические И. Если слева и справа true, то и результат true. В остальных случаях false.	true && true = true true && false = false false && true = false false && false = false;
||	Логическое ИЛИ.Если хотя бы один из операндов	true || true = true true || false = true

	true, то и результат true.	false || true = true false || false = false;
!	Логическое НЕ.Инвертирует значение.	!true = false !false = true;

Их можно также собирать как и арифметические операторы. Например:

(true && (false || true)) == true;

Если уйти от конкретных значений к абстракциям, можно перевести это выражение на человеческий язык:

Я открою это, если у меня есть ключ и это дверь или сундук.

Операторы отношений

Очень часто нам необходимо проверять отношения между значениями. В качестве операндов могут быть самые разные типы, а вот результатом всегда будет тип bool. Стандартные операторы формируют следующий набор.

Оператор	Описание	Пример
==	Равенство двух значений.	10 == 10 = true 10 == 15 = false true == true = true true == false = false;

!=	Неравенство двух значений.	10 != 10 = false 10 != 15 = true true != true = false true != false = true;

>, <, <=, >=	Больше, меньше, меньше или равно, больше или равно.	10 > 15 = false 10 < 15 = true 10 > 10 = false 10 < 10 = false 10 >= 10 = true 11 >= 10 = true 9 >= 10 = false 10 <= 10 = true

Как и другие операторы, эти операторы могут работать по-разному в отношении к разным типам. Так, например, мы можем создать сложный тип, который описывает армию игрока. И оператор меньше или больше будет сравнивать общую силу армии.

Переменные

Переменная – это одна из первых абстракций, с которой вам нужно научиться работать. Она позволяет записывать выражения в отрыве от конкретных значений, задавая только правило, что значение вписывается в определенный тип. Также переменные позволяют писать истинно императивные программы, основываясь на состояниях.

В переменной мы можем сохранить какое-то значение. Также в C# переменная на всю жизнь привязывается к определённому типу и, впоследствии, не может его поменять. Но, как понятно из названия, они вполне могут менять само значение.

В результате мы сможем записывать выражения по типу этого:

UserMoney = UserMoney – ItemPrice;
Что будет означать “убрать из кошелька пользователя сумму равную цене товара, который он приобретает”. Благодаря переменным мы уже можем записать несколько идущих подряд выражений, которые связаны между собой обсуждаемой сущностью.

Но, перед тем, как мы попробуем это сделать, давайте узнаем несколько правил. Во-первых, для объявления переменной нам нужно указать:

1. Тип;
2. Имя;
3. Первоначальное значение (не обязательно, но фактически её не будет существовать, пока мы не присвоим ей значение);

Выглядит это примерно таким образом:

```
int age = 21;
string name = “Roma”;
bool isInRelationship = true;
```

Слева указывается тип. Далее идет имя, которое также должно соблюдать несколько правил:

1. Не начинается с цифры (внутри цифру содержать может);
2. Нет пробелов и знаков пунктуации;
3. Не является ключевым словом;

При работе с переменными стоит узнать о новом операторе. Он называется “оператор присвоения” и выглядит как одинарное равно. Его семантика заключается в том, что он присваивает переменной слева значение выражения справа. При этом, мы можем воспринимать выражение как просто взятие текущего значения переменной.

Например:
int a = 10;
int b = a;

Т.е мы можем присвоить одной переменной значений другой.

Как я уже говорил, переменные очень важны для нас при описании выражений. Мы можем строить выражения на основе переменных, что даёт нам более абстрактные высказывания, которые мы можем применять в различных ситуациях.

Например:
bool isDoorOpen = age >= 18;

Мы создаём переменную isDoorOpen, которая будет равняться true, если возраст больше или равен 18.

Операторы управления

То, что мы рассматривали до этой главы, было в рамках одного выражения. Т.е вся программа была собрана в одно большое выражение и не имела сложную структуру выполнения.

Мы можем рассмотреть простую программу и понять, что ей кое-чего не хватает. Например:

Фрагмент 1.1

```
int money = 1000;
int itemPrice = 100;
money = money - itemPrice;
Console.WriteLine("После покупки у вас
осталось - " + money + " Р.");
```

Эта программа покупает предмет и пишет остаток. Это обычный последовательный алгоритм, состоящий из четырех шагов.

1. Инициализировать кошелек 1000 уе;
2. Инициализировать цену предмета 100 уе;
3. Отнять от значения кошелька цену предмета и присвоить результат кошельку;
4. Вывести остаток кошелька;

В некоторых случаях этого достаточно, но зачастую нет. Например, мы хотим добавить в программу:

1. Выбор предмета для покупки;

2. Вопрос с подтверждением покупки;
3. Проверку что у пользователя есть деньги;

К сожалению, в рамках операторов выражений всё это оптимально сделать крайне сложно.

Прежде чем мы перейдем дальше, я хотел бы ещё рассказать о некоторых правилах последовательной записи инструкций. Фактически исполняемый код мы можем в C# записывать в рамках функциональных членов, например, в методах, и как конкретная ситуация – в методе Main, который создаётся автоматически при создании консольного приложения.

Внутри функциональных членов мы можем перечислять последовательно выполняемые инструкции через точку с запятой. Также стоит учитывать правило хорошего тона, и ставить инструкции на новой строке.

Ещё стоит понимать, что мы не можем просто размещать выражения. Строки кода могут содержать выражения, но они должны быть связаны с вызовом функции или присвоением в переменную значения.

Структурная парадигма

Если говорить про текущую запись алгоритмов, то мы используем одну управляющую структуру – последовательность. Она позволяет нам переходить от одной инструкции к другой. Такой подход был раньше очень распространен, до того как в конце 60-ых годов мир программирования не изменился под влияниям одного из интереснейших формализмов.

Если говорить вкратце, то пришла идея отказаться от оператора goto, семантика которого заключается в том, что он позволяет перемещаться к любой строке кода. С помощью него можно было представить различную логику программы. Если вы когда-нибудь писали на ассемблер, то помните присутствие там оператора JMP, который передавал управление к некоторой строке кода. Он бывает в условной форме, т.е если некое значение чему-то ровно, то тогда происходит переход. Что позволяло организовывать ветвление.

На основе оператора JMP или goto можно реализовать любую программу. Но по мере роста кодовой базы мы столкнемся со сложностями управления. К сожалению программа будет тяжело читаема и сложна в поддержке. Взамен спонтанным переходам было предложено более чётко

декларировать блоки кода с помощью введения более строгих управляющих структур.
Всего есть три классических управляющих структуры:

1. Последовательность – сначала А потом В;
2. Ветвление – если Р то А если нет то В;
3. Цикл – делать В пока Р ;

Из этой парадигмы мы можем взять также несколько принципов разработки, которые немного изменяются, например, в объектно-ориентированной парадигме, но в целом очень мудрые и позволят вам лучше разрабатывать код.

1. Принцип. Отказаться от оператора безусловного перехода goto;
2. Принцип. Любая программа состоит из трех управляющих структур: последовательность, ветвление и цикл;
3. Принцип. Все три структуры могут быть вложены друг в друга в произвольном порядке. Например, цикл в цикле;
4. Принцип. Дублирующийся код должен быть оформлен в функцию;
5. Принцип. Нужно осознавать группы логически связанных инструкций и заключать их в блоки;
6. Принцип. Логические законченные блоки мы должны рассматривать с одним входом и одним выходом;
7. Принцип. Разработку нужно вести сверху вниз;

Эти семь принципов для опытного разработчика звучат довольно очевидно. И многие даже не читали их, а просто опытным путем дошли до этого. Но вот начинающим разработчикам нужно пройти через множество задач, чтобы понять описанное здесь. К сожалению, эта книга не про это.

Тут вы не встретите подробного объяснения этих принципов. Далее мы разберём проявление этой парадигмы в языке C#. В дальнейшем в заданных разделах я часто буду показывать свои решения, и в них вы сможете увидеть влияние этой парадигмы и сделать для себя выводы.

Область видимости

Эти управляющие конструкции организуют так называемые блоки кода, которые могут собираться в иерархии. Не вдаваясь сейчас в подробности синтаксиса, приведу пример. Если вы немного знакомы с английским, вам сразу станет понятней.

Фрагмент 1.2

```
int money = 1000;
if(money >= 1000)
{

}
```

Блочный оператор if создал блок с помощью фигурных скобок. Внутри этого блока своя область видимости, которая расширяет родительскую область. Т.е область заданную, например, методом или другим блочным оператором, в который вложен этот if.

Внутри этого if доступно всё, что доступно из родительской области плюс то, что определено внутри этого блока, но не в дочерних блоках.

Фрагмент 1.3

```
int money = 1000;
```

```
if(money >= 1000)
{
    int age = 18;
    Console.WriteLine($"{age} {money}");
}
```

Т.е внутри if доступна и переменная age и переменная money. Давайте попробуем определить ещё один блок кода.

Фрагмент 1.4

```
int money = 1000;
if(money >= 1000)
{
    int age = 18;
    if(age > 16)
    {
        bool access = true;
    }
    Console.WriteLine($"{age} {money}
{access}");
}
```

В такой программе содержится ошибка. Дело в том, что внутри if который проверяет, что age больше 16, создаётся переменная, недоступная вне этой области.

Т.е за пределами его фигурных скобок, переменная access перестаёт существовать. Это имеет смысл хотя бы потому, что если возраст пользователя меньше 16, то мы бы и не зашли в этот блок, что создало бы опасную ситуацию.

Если хочется изменять переменную во вложенных блоках, то её определение можно поднять выше, оставив изменение внутри.

Фрагмент 1.5

```
int money = 1000;
if(money >= 1000)
{
    int age = 18;
    bool access = false;
    if(age > 16)
    {
        access = true;
    }
    Console.WriteLine($"{age} {money}
{access}");
}
```

Условные операторы

Вы уже понимаете, что операторы могут создавать свои блоки кода, которые образуют область видимости. Нам осталось малое – узнать об операторах, которые мы можем использовать.

В C# есть три условных оператора, которые нас сейчас интересуют: if-elseif-else, switch и тернарный оператор ?:. Первые и второй имеют как блочную, так и сокращённую форму (на самом деле она тоже образует блок), а третий имеет больше оператор выражений, хоть с помощью него и можно организовать полноценное ветвление программы.

Мы также можем найти другие операторы, которые имеют немного другое применение. Но обсуждать мы их не будем.

If

С помощью оператора if мы можем задать блок кода, который выполнится при соблюдении определённых условий. Его применение вы могли увидеть немного выше. Он состоит из двух основных элементов: выражения и инструкций.

Его суть крайне проста – если результат выражения true, то выполняется блок кода. Если нет, то он

пропускается. Оператор if как и все блочные, не обрывают свой родительский блок, а только создают вариант возможных событий.

Проследите за примером.

Фрагмент 1.6

```
Random random = new Random();
Console.WriteLine("Tier1");
if(random.Next(0, 2) == 1)
{
    Console.WriteLine("Tier2");
}
Console.WriteLine("Tier3");
```

В нём мы используем рандом при составление выражения оператора if, в результате каждого запуска программы мы будем видеть либо три сообщения либо два. Дело в том, что есть шанс, что не выполнится блок оператора, но точно будут выполнены операторы до и после.

If и логические операторы

Более точно оператор можно описать таким образом:

```
If( ВЫРАЖЕНИЕ )
{
  ОПЕРАТОРЫ;
}
```

В качестве выражения могут выступать любые выражения, результатом которых будет bool. Простейшим случаем будет использование одного из операторов отношений. Как вы уже знаете, у нас в арсенале также есть ряд логических операторов, которые позволяют описывать более сложные выражения, нежели с использованием только операторов отношений.

Представим, что у нас есть небольшая игра, в которой персонаж перемещается по клеткам глобальной карты и при столкновение с врагом, перед боем, мы должны вывести шанс победы игрока над противником.

Фрагмент 1.7

```
int playerPositionX = 5;
int playerPositionY = 8;
int playerHealth = 100;
int playerDamagePerStep = 25;

int enemyPositionX = 5;
int enemyPositionY = 8;
int enemyHealth = 140;
```

```
int enemyDamagePerStep = 5;

if(playerPositionX == enemyPositionX &&
playerPositionY == enemyPositionY)
{
    int playerStepsToDeath = playerHealth /
enemyDamagePerStep;
    int enemyStepsToDeath = enemyHealth /
playerDamagePerStep;

    if(playerStepsToDeath >
enemyStepsToDeath)
    {
        Console.WriteLine("Вы скорее всего
победите");
    }
}
```

В первом операторе мы используем два оператора сравнения, результаты которых комбинируем с помощью логического И. В результате блок выполнится только, если мы стоим на одной клетке с противником.

Далее мы используем еще один оператор, который предсказывает сможет ли игрок победить противника, и если да, то выводит сообщение.

If-else-else if

Давайте представим, что мы хотим добавить в предыдущий пример условный “А что если нет?”. Допустим если игрок может выиграть, то мы выводим определенное сообщение. А что если нет? Ответ на этот вопрос мы можем дать с помощью оператора else.

Важно понять, что операторы после оператора if и в случае выполнения и в случае невыполнения всё равно выполняются. А вот операторы в блоке “А что если нет?” выполняются ТОЛЬКО ЕСЛИ if НЕ ВЫПОЛНЯЕТСЯ.

Например:

Фрагмент 1.8

```
string password = "QWERTY";
string input = Console.ReadLine();

if(input == password)
{
    Console.WriteLine("Добро пожаловать!");
}
else
{
    Console.WriteLine("Пароль не
правильный!");
}
```

В данном случае мы используем оператор else (C# синоним “А что если нет?”), благодаря которому мы можем что-то сделать если пароль НЕ ПОДОШЕЛ. Я думаю, вы уже понимаете, что оператор else идёт рука об руку с оператором if и является его продолжением, нежели самостоятельным оператором.

Т.е вы не можете записать просто оператор else, перед ним должен идти if.

Иногда, нам нужно расширить наше “А что если нет, но при этом”.

Например:

Фрагмент 1.9

```
string password = "QWERTY";
string adminPassword = "ADMINRODIP";
string input = Console.ReadLine();

if(input == password)
{
    Console.WriteLine("Добро пожаловать!");
}
else if(input == adminPassword)
{
    Console.WriteLine("Включен
```

```
административный режим!");
}
else
{
    Console.WriteLine("Пароль не
правильный!");
}
```

В данном примере, если не срабатывает основной оператор if, мы переходим к else if, который сработает, если в нем верное выражение. Оператор else if может быть сколько угодно, а также они могут оканчиваться единичным блоком else (а ещё его можно не делать).

Я предполагаю, у вас может возникнуть вопрос: “Какая разница между if-else if и двумя if?”. Ответ прост – два ифа могут выполняться друг за другом, а в случае с if-else if, блок else if выполнится, если не выполняется основной if.

Операторы циклов

Циклы нам нужны для того, чтобы осуществить повторное выполнение действий. Сила операторов циклов в том, что мы можем задавать количество выполнений с помощью переменных, что означает, что программа может гибко адаптироваться к внешним условиям.

Понятие цикла близко к понятию рекурсивной функции. И вправду многие циклы мы можем выразительно представить через рекурсию и рекуррентные отношения. Но цикл сам по себе более императивная вещь, которая буквально пытается представить алгоритм.

В C# мы имеем несколько операторов для представления циклов:

1. while, do-while – нужны для повторения некоторых действий без четкого понимания, сколько действий потребуется. Есть только условие продолжения;
2. for – нужен для циклов у которого заранее известно сколько итераций (выполнений) нам нужно сделать;
3. foreach – более высокоуровневый цикл, который является встроенным в C# оператором для работы с некой сущностью “итератор”. При работе с ним мы перебираем

некое множество элементов. Его мы рассмотрим в разделе с массивами и интерфейсом IEnumerable.

while

Простейший цикл, внешне схож с оператором if, но имеет другую семантику.

У него также есть условие и тело. Только в отличие от оператора if мы немного по-другому управляем программой. В случае с оператором if мы имеем условие и тело, которое выполняется, если условие верное.

Далее в любом случае выполняются операторы после оператора if. А с циклом while есть одно интересное дополнение. Если условие верно, то после выполнения тела, мы опять возвращаемся к условию, если условие все еще верно то всё повторяется.

Цикл мог бы быть бесполезным, но внутри его тела мы можем изменить обстоятельства таким образом, чтобы условие перестало быть верным и мы бы проследовали дальше. Чаще всего в условие мы используем переменную или вызов метода, состояние объекта которого может изменяться во время итераций.

Фрагмент 1.8

```
int secretNumber = 250;
int userInput = 0;

while(userInput != secretNumber)
{
    Console.WriteLine("Введите секретное
число:");
    userInput =
Convert.ToInt32(Console.ReadLine());
}

Console.WriteLine("Вы ввели правильное
секретное число!!!");
```

Если заменить в этом примере while на if, то программа скомпилируется и будет работать. Только у пользователя будет одна попытка ввести секретное число. И в любом случае программа выведет, что число было правильным. ¯_(ツ)_/¯

for

Любой цикл можно описать с помощью цикла while. Даже цикл foreach в процессе трансляции преобразуется в цикл while. Но как было и с оператором goto, нам недостаточно только возможности. Нам нужен ещё и удобный уровень абстракции и специализированные инструменты.

Одним из таких инструментов является цикл for. Он невероятно удобен при итерировании (переборе) массивов, а также выполнении операций, число которых можно описать.

Его синтаксис выражает три основных блока, которые присущи подобным циклам:

1. Инициализация счётчика;
2. Условие;
3. Изменение состояния (обычно инкремент);

Эти блоки мы перечисляем внутри круглых скобок оператора for через точку с запятой. Мы также вполне можем оставить этот блок пустым. Выполняются они в таком порядке:

1. Инициализация;
2. Условие, если верное, идём на шаг 3, если не верное уходим от оператора for;
3. Тело цикла;
4. Изменение состояния ;
5. Переход на шаг 2;

Фрагмент 1.9

```
int userAge = 0;

for(int i = 0; i < userAge; i++)
{
    Console.WriteLine("Happy birthday!");
}
```

Данная программа поздравляет пользователя с днём рождения столько раз, сколько ему лет.

Вырожденные операторы

Бывают ситуации которые стоит избегать. Например когда тело оператора вырождается, это часто бывает с оператором while.

Бывает такой код некоторые пытаются уместить в одну строку.

```
int number = 2;
int delimeter = 2;

while(number % delimeter++ == 0) ;

Console.WriteLine(delimeter);
```

Данный алгоритм ищет ближайший делитель числа без остатка.

Записывать это в одну строку нет смысла только если вы не хотите запутать читателя или показать свой навык владения синтаксисом. Правильно будет так:

```
int number = 2;
int delimeter = 2;

while(number % delimeter == 0)
```

```
{
   delimeter++;
}

Console.WriteLine(delimeter);
```

Массивы

Массив – это, в первую очередь, структура данных. Её обычно разделяют на два вида, поэтому мы можем встретить такие определения как:

1. Статический массив – имеет фиксированное количество элементов и не может расширяться или сужаться.
2. Динамический массив – имеет динамический объём, т.е. может расширяться и сужаться в процессе выполнения.

Второе определение я считаю ошибочным, так как массив, по определению, это и есть статическая структура данных. В C# есть реализация статического массива, с помощью типа List<T>, но внутри него обыкновенный статический массив и расширение листа происходит за счёт создания нового массива и переноса в него старого.

Для того, чтобы это происходило быстро, внутри происходит так называемая амортизация. Суть амортизации, в данном случае, заключается в том, что расширение происходит не на один элемент, а на текущее количество элементов в старом массиве, умноженное на два.

Массив дает очень хорошие результаты на операцию чтения по индексу. Также он обычно довольно

неплох при осуществлении перебора. Но он очень неудобен тогда, когда мы работаем с некоторым переменным количеством элементов. Например, когда мы хотим записывать позиции игрока во время его перемещения по уровню.

Когда мы создаем массив, мы указываем тип его элементов. Условно каждый элемент определенного типа имеет размер S. В случае с ссылочными типами – мы храним только ссылки на объекты в памяти.

Во время создания массива в памяти компьютера выделяется количество памяти S * N + M. Где N – количество элементов, а M – метаинформация самого массива.

В итоге мы получим что-то типа этого:

Значение	10	4	15
Индекс	0	1	2
Адрес	0010	0018	0026
Размер элемента – 8	Начало массива по адресу – 0010	Длина – 3	

Каждый элемент имеет индекс, который начинается с нуля. В дальнейшем, при работе с массивом мы указываем индекс элемента, чтобы прочитать или изменить его значение.

Поиск элемента в массиве происходит очень быстро. Чтобы получить адрес в памяти значения под индексом 1 – нам нужно умножить этот индекс на размер элемента и прибавить результат к адресу начала массива. В итоге у нас получится адрес элемента в памяти.

1 * 8 + 0010 = 0018

Именно поэтому у массива достаточно дешевая операция чтения по индексу.

Чаще всего, при работе с неким массивом мы не знаем о его длине. Точнее, длина нам известна, но мы учитываем, что его длина может быть любой и мы производим действия над множеством элементов.

Это возможно благодаря наличию циклов. Счётчик цикла мы используем как индекс элемента, а верхнюю границу цикла берём за длину массива. В результате этого мы можем последовательно обрабатывать все данные в массиве последовательно.

Использование массивов

В C# есть свой синтаксис для работы с массивами. Для начала определимся с терминологией.

1. Тип массива – это тип который, представляет нам массив элементов. Именно к объекту этого типа мы применяем операцию индексации (взятие элемента по индексу).
2. Тип элементов массива – данный массив состоит из элементов одного типа. Этот тип и называется типом элементов массива.
3. Длина массива – это количество элементов в массиве.

В C# тип “массив” образуется из типа элементов и квадратных скобок. Этот тип мы можем использовать при определении переменных, а также везде, где от нас требуется какой-нибудь тип.

Фрагмент 1.10

```
string[] users = new string[10];
```

Следующий код создает массив с 10 элементами. По умолчанию, в данном случае, в каждой ячейке массива будет значение null (такое значение будет для всех ссылочных типов).

Над переменной users мы можем осуществлять следующие операции.

Фрагмент 1.11

```
string[] users = new string[10];

users[0] = "Roma"; //Запись в первый
элемент
users[9] = "Vlad"; //Запись в последний
элемент

Console.WriteLine(users.Length); //Чтение
длины, выведет 10

string currentUser = users[users.Length -
1]; // Чтение последнего элемента, мы также
могли бы использовать значение 0
```

Как вы видите, мы повсеместно используем квадратные скобки. Они являются оператором индексации, внутри которого мы можем писать значения типа int, а также, как вы понимаете, любые выражения, в результате которых будет int.

Фрагмент 1.12

```
string[] users = new string[10];

users[0] = "Roma";
```

```
users[9] = "Vlad";

Console.WriteLine(users[5 - 5]); //Выведет
Roma
```

Перебор массивов циклами

С помощью циклов мы можем перебирать массивы. Давайте попробуем просто вывести все значения массива.

Фрагмент 1.13

```
string[] users = new string[3];

users[0] = "Roma";
users[1] = "Vlad";
users[2] = "Igor";

Console.WriteLine("В системе есть следующие
пользователи:");

for(int i = 0; i < users.Length; i++)
{
      Console.WriteLine(users[i]);
}
```

Данный код выведет все имена в массиве.

Задачи для практики массивов

Задача 1.

Объявите переменную, которая хранит массив, тип элементов у которого string и поместите туда ссылку на новый массив с типом элементов string, размерностью в 5 элементов.

Далее, через оператор индексации, проинициализируйте каждый элемент.

Задача 2.

Напишите программу, которая выведет все элементы массива.

Фрагмент 1.14

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 15;
numbers[2] = 2;
```

Задача 3.

Напишите программу, которая выведет сумму всех элементов массива.

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 15;
numbers[2] = 2;
```

Задача 4.

Напишите программу, которая выведет максимальное число в массиве.

Фрагмент 1.15

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 15;
numbers[2] = 2;
```

Тонкости инициализации

При работе с массивами мы можем использовать синтаксис инициализаторов, которые позволяют задать данные массива удобным синтаксисом.

Фрагмент 1.17

```
int[] array = {10, 14, 52, 14}; //Создаст
массив 4 элементами, можно использовать
только в строке объявления переменной.
array = new int[] {5, 2}; //создаст массив 2
```

```
элементами, можно использовать везде

int[,] map = {
    {1, 1, 1},
    {1, 0, 1},
    {1, 1, 1}
};
```

Массив map является двумерным массивом, подробней о таком типе вы узнаете на следующей странице.

N-мерные массивы

В предыдущих страницах мы работали с одномерными массивами. Их можно визуализировать как строку значений. У каждого значения в строке есть свой порядковый номер. Иногда этого недостаточно, тогда мы можем сделать N мерные массивы. N мерные они потому, что у них индекс элемента состоит не из одного, а из N значений.

Например, двумерные массивы можно визуализировать как таблицу, тогда каждое значение будет находиться на какой-то строке в каком-то столбце. Также можно сказать, что мы создадим двумерное поле и каждый элемент будет иметь позицию по X и Y.

Для получения длины измерения можно использовать метод GetLength, который принимает номер измерения. Для двумерного массива 3×4, длина первого измерения будет равна трем, а второго четырем.

Фрагмент 1.16

```
int[,] map = new int[3, 3];
map[0, 0] = 1;
map[1, 0] = 1;
map[2, 0] = 1;
map[0, 1] = 1;
```

```
map[2, 1] = 1;
map[0, 2] = 1;
map[1, 2] = 1;
map[2, 2] = 1;

for(int x = 0; x < map.GetLength(0); x++)
{
    for(int y = 0;y < map.GetLength(1); y++)
    {
        Console.Write(map[x,y]);
    }

    Console.WriteLine();
}
```

В результате у нас будет примерно такая таблица (матрица).

1	1	1
1	0	1
1	1	1

Массивы массивов

Как я предполагаю, вы уже догадались, что в качестве типа элементов массива может быть тип массива. В таком случае мы получим массив массивов или кортеж.

Один из атрибутов N-мерных массивов – это их ровность. Т.е в каждой строке будет одинаковое количество столбцов. В случае с массивами массивов это может быть не так.

Фрагмент 1.17

```
string[][] street = new string[][]
{
    new string[] { "Mary", "Thomas" },
    new string[] { "Carl" },
    new string[] { "Bob", "Jhonatan", "Alex"
}
};


for(int i = 0; i < street.Length; i++)
{
    Console.WriteLine($"Дома {i}:");
    for(int p = 0; p < street[i].Length; p++)
    {
        Console.WriteLine(street[i][p]);
    }
```

```
}
```

Методы

При разработке программ мы, к сожалению, не можем оперировать в голове всем тем количеством инструкций, из которых состоят серьёзные системы. Есть довольно старое правило, что человек может одновременно оперировать 5-7 вещами, при этом масштаб этих вещей не очень важен. Также, стоит заметить, что в основе психики человека лежит абстрагирование и обобщение, которое позволяет нам адаптироваться к новым, не знакомым нам вещам.
Разношерстные на первый взгляд инструкции можно объединить по смыслу, который они несут вкупе. Эти объединения можно собрать в более высокоуровневые абстракции. И так можно продолжать сколько угодно, пока вы не разобьете свою программу на иерархию понятных и простых в использовании сущностей.

Переход от буквального к семантическому очень важен, и чем более абстрактно программист может смотреть на происходящие в коде вещи, тем эффективней он может бороться с нарастающей при разработке сложностью системы.

Абстрагирование – это выделение существенных деталей на фоне несущественных. Например, выводя цветное сообщение в консоль, нам важно будет

только какого цвета сообщение и его текст. Это один уровень абстракции, на этом уровне нам не важно как внутри этой операции (вывод цветного сообщения) происходит возвращение к исходному цвету.

При этом мы можем рассматривать другой уровень абстракции. Если у нас есть операция вывода какого-то цветного сообщения, мы можем построить на её основе более высокоуровневую операцию – вывод предупреждения в консоль. Это операции выводит некоторый текст сообщения, желтым цветом и в рамке. При использовании этой операции неважно как там создается рамка и прочее, главное тут – её семантика – предупредить пользователя о чём-то.

Любая абстракция начинается с имени. Хотя она начинается еще раньше, в нематериальном плане идей и смыслов, но своё физическое воплощение, через которое мы передаем её другим людям, она начинает именно с имени. Хорошее имя – это лишь отражение хорошей сути. Если абстракция плохая, то как не думайте, будет у вас какой-нибудь Manager.

Метод как функциональная абстракция – метод есть некое действие, совершаемое с какими-то возможными входными значениями (аргументами), которое может завершится с каким-то результатом

(возвращаемое значение) и некоторыми корректными последствиями на состояние контекста (инварианты). Он как и любая абстракция описывает общую концепцию, а не конкретные примеры использования.

Мы можем как из лего строить одни функции из других тем самым создавая иерархию функций (алгоритмов), и в итоге, получая программу решающую нашу задачу эффективным и понятным читателю программы способом.
Если рассматривать метод как некий алгоритм или даже возможно действие, то мы можем говорить о двух важных аспектах: **предусловия и постусловия**.

1. Для выполнения действия нужны какие-то данные, инструкции или особые указания. Например если говорить, что у нас есть некое действия в системе как “Покупка товара” то такому действию нужно знать как минимум о том, кто покупает и что покупает.
2. У действия могут быть последствия. Например последствие покупки товара, информация об успешной или неудачной покупки + изменения баланса пользователя. Это есть постусловие.

При соблюдении предусловий, метод обязуется соблюдать свои постусловия. Это эмпирически понятно и обычно описывается словом “Контракт”.

Если вы дадите какой-то операции некое A, то она точно отдаст некое B.

В C# мы используем такой формализм как “формальная система типов” который позволяет описывать предусловия и постусловия с помощью декларации типов. Так предусловия мы можем выразить через типы параметров а постусловие через тип возвращаемого значения. Хоть это и не даёт возможности полностью следовать идеологии контрактного программирования, но это лучше чем ничего.

Спуск на один шаг

Имя функции должно находится на неком уровне абстракции N, весь код внутри функции должен располагаться чётко на уровне N – 1.

Пример когда функция уходит слишком вниз в своей реализации:

```
private static string MakeHtmlReport(Report
report)
{
    string fileBasePath =
Server.Request.ApplicationRunningPath;
    string html = $"<img
src='{fileBasePath}/report_logo.png'/>";
    html += $"<div>{report.Text}</div>";
}
```

Во-первых этой функции абсолютно не нужно знать о путях, во вторых функции очень нежелательно вручную собирать html.

Как бы выглядела эта функция если её реализации находилось бы на уровне ниже? Представим что на высоком уровне у нас Отчёт, а ниже находится абстракция “Ресурс” и “Формат”.

```
private static string MakeHtmlReport(Report
report)
```

```
{
      var page = htmlFormat.MakePage();
      page.Append(Resources.ReportLogo);
      page.Append(report.Text);
      return page.ToString();
}
```

Теперь в функции осталось самое важное выбор формата и передача через него данных. Но можно двинутся и дальше, например представить функцию MakeHtmlReport часть некоего полиморфного типа и у нас останется только это.

```
protected override string Make(Report
report)
{
       var page = _format.MakePage();
       page.Append(Resources.ReportLogo);
       page.Append(report.Text);
       return page.ToString();
}
```

У нас в сущности останется только шаблон и мы введём такую абстракцию в системе “Шаблон отчёта” и в результате будет что-то такое.

```
protected override void Draw(Report report,
IFormatedPage page)
{
        page.Append(Resources.ReportLogo);
        page.Append(report.Text);
}
```

Синтаксис методов

При создание первой консольный программы на C# в Visual Studio вы увидите простой каркас.

Фрагмент 1.18

```
class Program
{
    static void Main(string[] args)
    {

    }
}
```

Это программа содержащая один единственной метод Main и класс к которому этот метод относится Program. Понятие классов будет рассмотрено в следующей главе. Пока просто запомните что любой метод должен быть описан внутри какого-либо класса.

Этот метод является точкой входа в программу, именно из него вызываются основные методы нашей программы, которые в свою очередь вызывают другие методы и т.д. Технически, метод – это переход к какому-то именованному блоку инструкций, а потом

возврат из него. Это я наглядно покажу после того, как мы освоим базовый синтаксис.

Вызов метода

При вызове метода мы переходим в его тело и начинаем выполнять вложенные в него инструкции. Для вызова нам нужно применить к его имени оператор вызова, который представляется круглыми скобками. Внутри этого оператора через запятую мы перечисляем аргументы метода, те данные которые он от нас требует.

Например для вывода сообщения в консоль мы можем использовать метод Console.WriteLine который принимает данные для вывода.

Фрагмент 1.19

```
Console.WriteLine("Hello World");
```

В данном случае мы его вызвали и передали строку “Hello World” в качестве аргумента. Так как круглые скобки это оператор, в конце строки с ними должна стоять точка запятой.

Помимо этого, метод может возвращать данные, которые в языковых конструкциях воспринимаются

как результат выполнения оператора вызова. Так что вызов метода можно встретить в самых неожиданных местах. Например у нас есть метод Console.ReadLine который ничего не принимает (одна из перегрузок), но возвращает строку введенную пользователям в консоли. Он забирает управление, считывает символы которые пользователь вводит и возвращает управление с получившейся строкой после нажатия на enter.

Фрагмент 1.20

```
Console.WriteLine("Введите имя:");
string name = Console.ReadLine();
```

Как вы понимаете методы могут также принимать и возвращать значения в одном вызове. Например метод возведения числа в степень.

Фрагмент 1.21

```
double result = Math.Pow(10, 2);
```

Возводим число 10 во вторую степень.

Объявление метода

Это всё не имело бы смысла если бы не могли бы определять свои методы.

Выглядит это следующим образом

Модификатор ТипВозвращаемогоЗначения Имя (ТипПараметра ИмяПараметра,)

{

тело метода

}

- Под модификаторами подразумевается широкий спектр ключевых слов большую часть из которых мы рассмотрим далее. Пока что мы всегда будем указывать private static.
- Тип возвращаемого значения – это тип явного результата метода. Мы можем указывать любой доступный тип. Например если мы указываем int, то это означает что результатом работы метода будет какое-то число.

- Через имя метода мы его обычно вызываем.
- Список параметров нужен для указания того, что методу нужно. Если указываем через запятую два параметра с типом int то это значит, что при вызове метода, ему должны быть переданы два числа. Имена параметров нужны для оперирования ими в теле метода.

Для примера давайте определим метод, который возвращает количество повторений определенного символа в тексте.

Фрагмент 1.22

```
static int GetSymbolCount(string text, char
symbol)
{
    int count = 0;

    for(int i = 0; i < text.Length; i++)
    {
        if(text[i] == symbol)
        {
            count++;
        }
    }
```

```
    return count;
}
```

Здесь есть ряд интересных моментов

1. Мы можем использовать строку как массив символов (значений с типом char).
2. Если у метода тип возвращаемого значения не void то он должен обязательно вернуть какое-то значение этого типа при любом исходе. Возвращение значения (и прерывания выполнения метода) происходит с помощью инструкции return.

Воспользоваться им мы можем вот так

Фрагмент 1.23

```
int aCount = GetSymbolCount("Roma said that programming is easy", 'a');
```

Вызов метода – это переход в начало его тела и превращение параметров в локальные переменные

с инициализацией их значениями соответствующих аргументов.

Метод выполняет до своего конца или до инструкции return. После выполнения мы возвращаемся в точку его вызова. Если мы укажем в качестве возвращаемого значения void, то можем не использовать return, а если будем использовать то должны это делать без значения так, как указали что он ничего не возвращает.

Поток выполнения

При вызове метода мы перемещаемся в начало его тела, а инструкция return возвращает нас в то место, где был этот вызов. Точек вызова метода может быть много, и возвращаемся мы именно в ту точку, из которой произошел именно этот вызов.

Попробуйте проследить за этой программой и вычислить что будет выведено в консоль.

Фрагмент 1.24

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main");
        Method2();
```

```
        Method3();
        Method1();
        Console.WriteLine("End");
    }

    private static void Method1()
    {
        Console.WriteLine("Method1");
        Method2();
    }

    private static void Method2()
    {
        Console.WriteLine("Method2");
        Method3();
    }

    private static void Method3()
    {
        Console.WriteLine("Method3");
    }
}
```

Для некоторых будет удивительно, но выведет следующее:

Main

Method2

Method3

Method3

Method1

Method2

Method3

End

Модификаторы параметров ref и out

В C# любой тип является либо ссылочным, либо значимым типом. При работе со значением ссылочного типа, мы фактически всегда работаем с ссылкой на это значение. А в случае с типом значения мы работаем напрямую со значением. Отсюда следует что оператор присвоения копирует ссылку на значение для ссылочного типа и само значение для значимого.

Например int – это значимый тип. А любой тип массива – ссылочный.

Что выводится в консоль?

Фрагмент 1.25

```
int a = 0;
int b = a;
b = 10;
a = 15;

Console.WriteLine(a);
Console.WriteLine(b);
```

Правильный ответ: 10 и 15. И в этом нет ничего странного так, как в случае с типом значения (которым является тип int) при присвоение одной переменной другой, копируется значение

переменной и в данном случае это само значения. Соответственно в один момент переменная b и a имеют одинаковое значение но это разные экземпляры одного значения так, что изменение одной переменной не приводит к изменению другой.

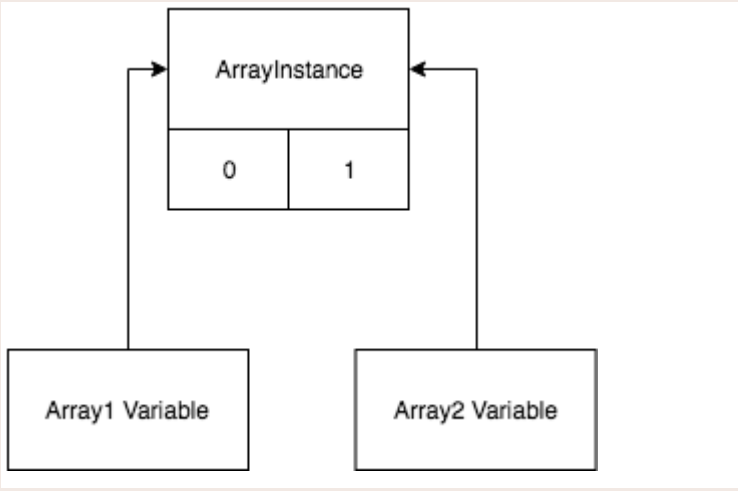
А что будет в этом случае?

Фрагмент 1.26

```
int[] array1 = new int[] { 0, 0 };
int[] array2 = array1;

array2[0] = 10;
array1[1] = 15;


Console.WriteLine(array1[0]);
Console.WriteLine(array1[1]);

Console.WriteLine(array2[0]);
Console.WriteLine(array2[1]);
```

Правильный ответ: 10 15 10 15

Переменные, с типом массива, содержат в себе ссылку на значение и при присваивании одной переменной к значению другой, мы копируем ссылку на один и тот же массив в памяти.



Всё это справедливо и для вызовов методов. Когда мы указываем переменную в качестве аргумента метода, мы просто берём значение переменной и передаем ее в метод.

Как думаете, что будет в консоли?

Фрагмент 1.28

```
static void Main(string[] args)
{
    int a = 15;
    int b = 10;
    int c = 0;

    Sum(a, b, c);

    Console.WriteLine(c);
```

```
}

static void Sum(int a, int b, int c)
{
    c= a + b;
}
```

Ответ ноль. Во-первых переменная “c” в методе Main и параметр “c” метода Sum никак не связаны. Во-вторых мы передаем в метод копию значения переменной c.

Также хочу заметить что если вам нужно узнать в точке вызова результат работы метода, то для этого существует возвращаемое значение.

Фрагмент 1.29

```
static void Main(string[] args)
{
    int a = 15;
    int b = 10;
    int c = 0;

    c = Sum(a, b);

    Console.WriteLine(c);
}

static int Sum(int a, int b)
```

```
{
    return a + b;
}
```

Идём далее. А как вы думаете что будет если передать массив в метод? Так как он ссылочный, передаваться будет копия ссылки на массив в памяти. Ну и контрольный вопрос. Что мы увидим в консоли?

Фрагмент 1.30

```
static void Main(string[] args)
{
    char[] map = new char[] { '_', '_', '!',
'_', '_', '_'};

    Generate(map);

    for (int i = 0; i < map.Length; i++)
    {
        Console.Write(map[i]);
    }
}

static void Generate(char[] map)
{
    for(int i = 0; i < map.Length; i++)
    {
        map[i] = '#';
```

```
    }
}
```

Как не странно линию из решёток.

Розы гибнут на газонах
А шпана на красных зонах

У вас должен возникнуть вполне резонный вопрос. Зачем я об этом рассказываю? В заголовке же написано “Модификаторы ref и out”. А ответ простой, весь этот текст рассказывает о том, что в метод нельзя передавать переменную. Что в него передается копия значения переменной.

И тут я такой хоба, достаю из широких штанин модификаторы которые… позволяют передавать ссылку на переменную и изменять её внутри метода. Давайте вспомним наш предыдущий пример с методом Sum и переменной “c” и попробуем выразить его таким образом, чтобы параметр “c” реально был привязан.

Фрагмент 1.31

```
static void Main(string[] args)
{
    int a = 15;
    int b = 10;
    int c = 0;

    Sum(a, b, ref c);
```

```
    Console.WriteLine(c);
}

static void Sum(int a, int b, ref int c)
{
    c = a + b;
}
```

При таком исполнение мы получаем действительно значение 25 в консоли потому, что указали что параметр “c” – это ссылка на какую-то переменную. И при вызове метода нам нужно указать какую-то переменную с тем же модификатором в качестве аргумента. После этого внутри метода изменяя параметр “c” мы изменяем внешнюю переменную.

Важно заметить что имя переменной и параметра может не совпадать.

Помимо модификатор ref у нас так же есть out. Технически они работают одинаково, но разность у них семантическая. Так ref говорит только о том, что будет ссылка на некую переменную которая должна иметь значение до вызова. А out говорит что переменная может быть пустая и метод обязан задать ей значение.

Например в методе Sum лучше было бы использовать out так, как метод Sum гарантирует, что в результате своего выполнения связанная переменная будет проинициализирована, а также ему не требуется какое-то изначальное значение переданной переменной.

Фрагмент 1.32

```
static void Main(string[] args)
{
    int a = 15;
    int b = 10;
    int c;

    Sum(a, b, out c);

    Console.WriteLine(c);
}

static void Sum(int a, int b, out int c)
{
    c = a + b;
}
```

Необязательные аргументы

Разрабатывая функцию, мы выводим абстракцию. То, как работать с абстракцией определяет её интерфейс, который представляется сигнатурой нашей функцией-метода. Сигнатура – это тип возвращаемого значения и список параметров.

Иногда методу нужны какие-то данные для работы и часть для них опциональна. Либо их может в принципе не быть, либо метод может представить значение по умолчанию. Для выражения таких ситуаций есть необязательные аргументы.

Фрагмент 1.33

```
static void Main(string[] args)
{
    Console.WriteLine("Warning!",
ConsoleColor.Yellow);
    Console.WriteLine("Error!",
ConsoleColor.Red);
}

static void WriteColoredMessage(string
message, ConsoleColor color)
{
    ConsoleColor oldColor =
Console.ForegroundColor;

    Console.ForegroundColor = color;
```

```
    Console.WriteLine(message);

    Console.ForegroundColor = oldColor;
}
```

В таком методе можно представить цвет как необязательное значение. И мы можем выразить это через специальный синтаксис.

Фрагмент 1.34

```
static void Main(string[] args)
{
    Console.WriteLine("Warning!",
ConsoleColor.Yellow);
    Console.WriteLine("Error!",
ConsoleColor.Red);
    Console.WriteLine("Empty message!");
}

static void WriteColoredMessage(string
message, ConsoleColor color =
ConsoleColor.White)
{
    ConsoleColor oldColor =
Console.ForegroundColor;

    Console.ForegroundColor = color;
    Console.WriteLine(message);
```

```
    Console.ForegroundColor = oldColor;
}
```

Хоть такой трюк иногда закладывается изначально, например, когда есть два пользователя одного метода, но им всем нужны немного разные наборы параметров.

Рассмотрим пример из Unity. Там есть метод Raycast, который кидает луч в физическом пространстве и сообщает нам о столкновении с ним. Сигнатура одной из перегрузок такая.

```
public static bool Raycast(Vector3 origin,
Vector3 direction, float maxDistance =
Mathf.Infinity, int layerMask =
DefaultRaycastLayers, QueryTriggerInteraction
queryTriggerInteraction =
QueryTriggerInteraction.UseGlobal);
```

Мы можем вызывать методы вот так. Рейкаст полетит с нулевых координат направо на расстоянии в 100 юнитов.

```
Physics.Raycast(Vector3.zero, Vector3.right,
100);
```

А можем, например, так, и он будет лететь с нуля направо в бесконечность.

```
Physics.Raycast(Vector3.zero, Vector3.right);
```

Но что делать, если нам нужен один из параметров, а другой не нужен? Например, я не хочу указывать дистанцию, но хочу указать маску слоя?

Я могу воспользоваться именованными аргументами. Я просто укажу то, кому предназначается этот аргумент.

```
Physics.Raycast(Vector3.zero, Vector3.right,
layerMask: _mask);
```

Здесь я в аргумент layerMask передал значение гипотетического поля _mask, при этом пропустив distance.

Эволюция API

При длительной разработке часто бывает, что метод уже написан и им активно пользуется, а также пользуется в тех местах, где код мы изменить не можем: какая-то часть команды, которая занята другим делом и которая использует наш код как библиотеку или мы просто поставляем свой код для других разработчиков.

В такой ситуации нам может понадобится расширить список параметров метода для введения какой-нибудь настройки. Если мы просто добавим параметр, то после обновления у себя нашей библиотеки разработчики столкнутся с ошибками. Это не очень приятно.

В таком случае мы можем просто добавить параметр в метод как необязательный и сделать ему значение по умолчанию, которое и было раньше.

В случае со значимыми типами, которые, раньше в принципе не имели определения в логике метода, мы можем воспользоваться nullable типами. Это может позволить нам определить: в метод всё-таки что-то передали или нам стоит работать будто ничего не происходило?

Функциональная композиция

Когда мы работаем с методами, они очень часто разрастаются. Просто добавление кода к уже имеющемуся коду в рамках одного метода приводит к его усложнению, и он начинает наполняться деталями, а его предназначение становится всё дальше и дальше для читателя.

В данном случае нам поможет движение по иерархии от общего к частному. Мы можем объединить некоторый код в отдельные методы\функции, сделав их ниже по отношению к вызываемому коду. Обычно, чем выше код (по стеку вызова), тем более общий он, и больше рассказывает о том, что или зачем делает программа, чем когда мы спустимся ниже, то мы будем действовать в ограниченном пространстве этого уровня. На таком уровне мы уже не контролируем как нами будут пользоваться, и не совсем понимаем зачем, нам лишь нужно выполнить наш контракт.

Сверху тоже обычно не важно как там всё работает в деталях, главное чтобы выполнили свой контракт.

Если в методе уже стало слишком много операций, то их стоит выделить в отдельные методы. Иногда это выражается тем, что программист с помощью пустых

линий располагает кусочки метода отдельно друг от друга – это уже подсказка, что пора выделять метод.

Выделение метода также даёт свои плоды: код, который мы выделяем становится более формальным, у него появляется чёткий контракт входа и контракт выхода. Его можно отдельно протестировать, и им легче оперировать.

Функциональная композиция может быть как благом, так и анти-паттерном (функциональная декомпозиция). Например, у нас в программе может появится класс работающий с файлом. Назовём его SaveLoader, он открывает файл и начинает самостоятельно работать со сценой, проставляя загруженные данные объектам. Корректно ли это? Чаще всего нет, так как в нём много других ответственностей. Хоть мы и разобьём его на композицию различных методов, все же эти методы будут расположены совершенно не там, из-за чего мы прибьём ОО дизайн, и с помощью функций раздекомпозируем систему. Подробней об этом мы поговорим в следующих главах.

А пока давайте подумаем над таким примером, и попробуем разбить его в рамках одного класса. Начав видеть методы, вы сможете начать учиться видеть, где эти методы должны располагаться.

```
private static bool TryOpenDoor()
{
    Console.WriteLine("Привет! Сколько тебе
лет?");
    var age = 0;

    while (int.TryParse(Console.ReadLine(),
out age) == false)
    {
        Console.WriteLine("Это не число!");
    }

    if (age <= 18)
    {
        Console.WriteLine("Дверь не для
тебя!");
        return false;
    }
    else
    {
        return true;
    }
}
```

Что мы можем сказать о нём? Во-первых этот метод сильно связан с консолью, во-вторых сам метод пестрит деталями. Его суть заключается в том, что дверь не открывается для тех, кому нет 18 лет. Как мы можем это представить?

Для начала давайте выделим метод чтения числа. Метод уже подсократился, не правда ли?

```
private static bool TryOpenDoor()
{
    Console.WriteLine("Привет! Сколько тебе
лет?");
    var age = ReadInt();

    if (age <= 18)
    {
        Console.WriteLine("Дверь не для
тебя!");
        return false;
    }
    else
    {
        return true;
    }
}

private static int ReadInt()
{
    int result = 0;
    while (int.TryParse(Console.ReadLine(),
out result) == false)
    {
        Console.WriteLine("Это не число!");
    }

    return result;
```

```
}
```

Делаем ещё шажок:

```
private static void InteractWithDoor()
{
    Console.WriteLine("Привет! Сколько тебе
лет?");
    var age = ReadInt();

    if (TryOpenDoor(age) == false)
    {
        Console.WriteLine("Дверь не для
тебя!");
    }
}

private static bool TryOpenDoor(int age)
{
    return age >= 18;
}

private static int ReadInt()
{
    int result = 0;
    while (int.TryParse(Console.ReadLine(),
out result) == false)
    {
        Console.WriteLine("Это не число!");
    }
```

```
    return result;
}
```

Правило программы, что двери открываются при достижении 18 лет, мы полностью очистили от другого кода. Всё взаимодействие с дверью мы локализовали в InteractWithDoor, он уже вызывает ReadInt и отделённый от текстовых разборок TryOpenDoor.

Мы сделали сущую мелочь, но она встречается повсеместно, и, если взять себе в привычку разбивать методы на составляющие и научится делать понятные имена, то ваш код преобразится!

Классы (ООП?)

Класс и объект

Если мы говорим про метод – мы говорим про функциональную абстракцию. В методах (функциях) мы заключаем алгоритм-операцию. Мы можем рассматривать функцию более обще и придавать ей дополнительные свойства, добиваясь своей цели. Но семантика нашего языка именно такая, что мы ограничиваемся только полем фактически выполняемых инструкций.

А что нам может понадобится ещё? При разработке программ нам может быть полезно думать в разрезе сущностей. Сущность – это объединение данных и операций над ними. Совокупность данных определяет некоторое состояние. Операции выполняются не только над аргументами, но и над состоянием контекста. Контекстом и является некоторая сущность.

Ранее я упомянул про чистые функции (Pure Function). Это функции, которые зависят и работают только о того, что им явно передаётся на вход. При работе с классами и объектами мы расширяем контекст функции до состояния объекта. Хорошая функция, как правило, работает с состоянием объекта, в котором она вызывается + с входными аргументами.

Чтобы работать с сущностями нам на самом-то деле классы не нужны. Даже скоп глобальных функций и некие куски глобального состояния мы можем рассматривать как те или иные сущности. А если сущности имеют тенденцию к размножению, т.е имеет несколько одновременно существующих экземпляров с разным состоянием, то нам не составит труда справится и с этим вызовом.

Но если мы прибегнем к синтаксису классов и объектов, нам станет гораздо легче жить. У класса есть несколько первичных определений, которые помогут понять его предназначение. Во-первых, класс – это новый тип данных, который мы вводим в систему. Во-вторых, так как это тип данных – он позволяет сформализировать систему и сделать её более надёжной. Как и в случае с обязательной декларацией параметров и их типов в методах.

Класс возникает тогда, когда мы начинаем осмысливать систему как набор взаимодействующих объектов. Когда мы можем объединить объекты по структуре, мы делаем класс. Это как с дублирующимся кодом. Даже если он отличается в значениях, но структура у него общая, то мы можем создать метод и разные значения подставлять через аргументы.

Например, у нас могут присутствовать пользователи в системе. Что из себя представляет каждый пользователь? Имя, зарплату в час и текущий невыплаченный остаток. В вашей голове пользователь представляет нечто другое? Вполне возможно. Очень ошибочно мнение, что классы и объекты в ООП – это отражения объектов реального мира. На самом деле – это, в первую очередь, описание объектов бизнес-логики. И если в нашем приложение нам нужно, чтобы у пользователя было имя и мы могли начислять ему деньги на основе отработанных часов, то структура у него будет именно такая.

То, что я описал – это члены данных. Мы описали данные, но не операции над ними. В качестве операций может служить операция “Сдать отчёт о работе”. В нём мы указываем отработанные часы и на основе них мы зачисляем пользователю невыплаченный остаток, учитывая его ставку в час.

То, что мы описали – это класс. Когда я говорю где-то в системе, что мне нужен пользователь, то этот пользователь будет именно такой структуры. Я смогу без опаски выполнять заданные операции и работать с доступными данными. А когда уже идёт непосредственная работа с пользователем, мы говорим про объект. Когда у нас уже есть конкретный

пользователь, с конкретным именем – это объект. А общее описание пользователя – это класс.

Члены и создание объектов

Если заниматься не только общим описанием этой темы, но и показать примеры, то нужно разобраться уже более конкретно: что и из чего состоит. Сейчас мы попробуем поработать уже с чем-то более осязаемым.

В рамках класса мы можем определять различные члены, которые будут являться частью объекта. На самом деле член может быть членом или класса или объекта. Разница в том, что когда член принадлежит всему классу – он доступен без создания объекта и как бы существует глобально. Такие члены называются статическими.

Если член определён в классе и не помечен как статический, то он является членом объекта. И обращаться к нему можно через экземпляр объекта. Нас интересует следующие члены:

- Поля
- Методы
- Конструкторы

Поля

Объект может содержать данные. Совокупность данных – это состояние объекта. Класс описывает что у каждого объекта должны быть определённые

данные, а в каждом конкретном объекте свои данные.

Так например мы можем сказать что у нас есть класс Tank описывающий танк в игре. У каждого танка есть скорость хода, которую мы описываем как поле. А каждый объект уже содержит конкретное значение скорости.

Поля – это переменные привязанные к объекту. Правила определения примерно похожи. Соответственно когда мы создаём несколько объектов, мы имеем несколько переменных.

Фрагмент 1.35

```
class Tank
{
    public int Speed;
}
```

В коде выше мы определили класс с одним полем Speed. Отличие от обычной переменной в том, что во-первых мы определили её в теле класса а не метода, а во-вторых в том, что мы указали модификатор доступа. Модификаторы доступа – это указание кому доступен этот член.

Часть объекта может быть доступна например всем желающим (публичные члены) и часть только самому объекту (приватные члены). Это необходимо для

достижения различных концепций. Подробно об этом будет далее.

После определения класса мы можем начать создавать объекты.

Фрагмент 1.36

```
static void Main(string[] args)
{
    Tank tank1 = new Tank();
    Tank tank2 = new Tank();

    tank1.Speed = 10;
    tank2.Speed = 15;
}
```

В данном фрагменте мы создали в памяти два объекта типа Tank и назначили каждому танку свою скорость. **Стоит упомянуть, что класс – это ссылочный тип.**

Фрагмент 1.37

```
Tank tank1 = new Tank();
Tank tank2 = new Tank();

tank1 = tank2;

tank1.Speed = 10;
tank2.Speed = 15;
```

```
Console.WriteLine(tank1.Speed);
```

В данном случае в консоль выведется 15, так как переменная с типом класса хранит ссылку на объект. В строке “tank1 = tank2;” мы помещаем в переменную tank1 ссылку на объект, которая содержится в переменной tank2. Таким образом при изменении объекта по ссылке из переменной tank2 мы также будем работать с изменённым объектом по ссылке в переменной tank1.

Также вы могли заметить, что доступ к членам объекта происходит через оператор точка. Имея выражение которое является ссылкой на объект, мы можем применить к результату оператор точку и получить доступ к доступным нам членам.

Фрагмент 1.38

```
static void Main(string[] args)
{
    (new Tank()).Speed = 10;
    CreateNewTank().Speed = 10;
}

static Tank CreateNewTank()
{
    return new Tank();
}
```

Методы

Поля описывают данные, а методы – операции над ними. На предыдущих страницах мы уже познакомились с тем, что такое методы. Но как вы могли заметить, в прошлый раз мы к каждому методу приписывали static.

Теперь мы этого делать не будем. На данный момент каждый метод будет членом объекта и в процессе выполнения он сможет изменять состояние своего объекта. Давайте вернёмся к примеру с танком.

Фрагмент 1.39

```
class Program
{
    private static void Main(string[] args)
    {
        Tank tank1 = new Tank();

        tank1.Speed = 10;

        tank1.MoveRight();
        tank1.MoveRight();

        Console.WriteLine(tank1.PositionX);
//20
    }
}
```

```
class Tank
{
    public int Speed;
    public int PositionX;

    public void MoveLeft()
    {
        PositionX -= Speed;
    }

    public void MoveRight()
    {
        PositionX += Speed;
    }
}
```

Мы добавили в танк его позицию по X оси. А также два метода, которые двигают его вправо или влево в зависимости от его скорости. Как вы видите, при вызове этих методов они выполняются в контексте объекта.

Т.е имеют доступ к позиции и скорости танка, у которого они были вызваны.
Давайте рассмотрим разницу между членом объекта, с которым он работает, и методом статическим.

Инкапсуляция

С помощью методов мы также можем достичь инкапсуляции. Давайте представим, что у объекта

появился ряд специфичных правил. Так, при движении вправо или влево мы должны следить за тем, чтобы позиция по X не была меньше нуля и больше 100.

Так, для того, чтобы это правило всегда соблюдалось, нам хотелось бы запретить изменять позицию танка напрямую. Т.е позиция может изменяться только через движение влево или вправо методов. Но для того, чтобы мы могли нарисовать танк в правильной позиции, нам нужно чтобы позицию могли читать.

Т.е мы закрываем позицию для записи, но оставляем открытой для чтения.

Это всё мы можем достичь с помощью методов.

Фрагмент 1.40

```
class Tank
{
    public int Speed;
    public int PositionX;

    public void MoveLeft()
    {
        PositionX -= Speed;

        if (PositionX < 0)
```

```
            PositionX = 0;
    }

    public void MoveRight()
    {
        PositionX += Speed;

        if (PositionX > 100)
            PositionX = 100;
    }
}
```

Мы добавили в методы движения проверку на выход позиции за границы. Но всё ещё можно изменить позицию напрямую, в обход наших методов, тем самым сломать состояние нашего объекта.

Фрагмент 1.41

```
class Program
{
    private static void Main(string[] args)
    {
        Tank tank1 = new Tank();

        tank1.Speed = 10;

        tank1.MoveRight();
        tank1.MoveRight();
```

```
Console.WriteLine(tank1.GetPositionX()); //20
    }
}

class Tank
{
    public int Speed;
    private int _positionX;

    public void MoveLeft()
    {
        _positionX -= Speed;

        if (_positionX < 0)
            _positionX = 0;
    }

    public void MoveRight()
    {
        _positionX += Speed;

        if (_positionX > 100)
            _positionX = 100;
    }

    public int GetPositionX()
    {
        return _positionX;
    }
}
```

И вот мы изменили поле PositionX. Во-первых, мы изменили модификатор доступа с public на private. В результате этого, поле теперь недоступно за пределами объекта. Т.е его могут изменять методы определённые в классе. Но методы определённые в других классах, имея ссылку на объект этого типа, уже не могут обращаться к этому члену (члену определённому с модификатором private).

Во-вторых, мы, согласно нотации, приватное поле назвали с маленькой буквы и с нижним подчёркиванием. Это не обязательно, но лично я применяю именно этот стиль программирования.

Так как поле не доступно вне класса вообще, то нам надо предоставить доступ на чтение. Это мы сделали с помощью метода GetPositionX. Он возвращает текущее значение поля и мы пользуемся этим методом, когда хотим узнать текущее положение танка.
Давайте подумаем про инкапсуляцию в таком контексте. Нам захотелось убрать из танка ответственность ограничения движения. Точнее делегировать определения возможных позиций другой сущности. Например.

```
class Boundary
{
    public bool IsAvailable(int positionX);
```

```
}
```

Мы не будем вдаваться в подробности содержания этого класса. Допустим он просто следит за тем, чтобы некая позиция была доступна если там нет стены и она в границах уровня. Промодифицируем класс танка.

```
class Tank
{
    public int Speed;

    private int _positionX;
    private Boundary _boundary;

    public Tank(Boundary boundary)
    {
       _boundary = boundary;
    }

    public void MoveLeft()
    {
        Move(Speed * -1);
    }

    public void MoveRight()
    {
       Move(Speed);
    }
```

```
    private void Move(int delta)
    {
        if (_boundary.IsAvailable(_positionX
+ delta))
            _positionX += delta;
    }

    public int GetPositionX()
    {
        return _positionX;
    }
}
```

Соблюдаем ли мы инкапсуляцию в таком случае? Допустим мы говорим: танк всегда находится в пределах уровня и не пересекается с другими объектами. Можем ли мы это гарантировать?

Можно сказать что да. Но на самом деле нет. Хоть и нельзя сменить Boundary, т.е в процессе выполнения конкретному танку заменить контекст в котором он находится чтобы его позиция стала не валидной, но можно изменить текущий контекст.

Если бы класс Boundary был изменяемый то тот, кто нам ссылку предоставил, мог изменить объект как ему хочется.

Подробней эта проблема инкапсуляции будет изложена в другой главе.

Конструкторы

Конструктор – это синтаксически выделенный метод для создания объектов. Если мы говорим про обычные методы – они могут вызываться с любом порядке и если нам нужно как-то инициализировать объект перед вызовом методов мы можем сделать метод Init. Но к сожалению никто не обязан его вызывать, и вообще необходимость его вызова может быть не очевидной из-за чего в программе может появится ряд неприятных ошибок.

Если нам нужно инициализировать объект мы можем воспользоваться конструктор. Он вызывается первым в момент создания объекта. Он как и метод может принимать параметры.

Если возвращаться к примеру с танком мы можем сказать, что при создание танка нам обязательно нужно указать его скорость и начальную координату.

Фрагмент 1.42

```
class Program
{
    private static void Main(string[] args)
    {
        Tank tank1 = new Tank(10, 0);

        tank1.MoveRight();
```

```
        tank1.MoveRight();


Console.WriteLine(tank1.GetPositionX()); //20
    }
}

class Tank
{
    public int Speed;
    private int _positionX;

    public Tank(int speed, int positionX)
    {
        Speed = speed;
        _positionX = positionX;
    }

    public void MoveLeft()
    {
        _positionX -= Speed;

        if (_positionX < 0)
            _positionX = 0;
    }

    public void MoveRight()
    {
        _positionX += Speed;

        if (_positionX > 100)
```

```
            _positionX = 100;
    }

    public int GetPositionX()
    {
        return _positionX;
    }
}
```

Как вы видите по этому фрагменту мы добавили странный метод в класс Tank. Он называется точно также и при этом не возвращает никакого значения.

А используем мы конструктор при использование оператора new. Как вы видите у нас есть следующая строка “Tank tank1 = new Tank(10, 0);”. Скобки по сути вызывают конструктор и передают ему необходимые аргументы.

Конструктор – это контракт объекта, он позволяет гарантировать что на момент вызова его методов будут разрешены зависимости и состояние объекта будет находится в корректной форме.

Связь объектов (Has-a)

Когда мы используем объекты мы обычно приходим к объектному графу. Со временем у нас появляется несколько объектов которые взаимодействуют друг с другом. Когда один объект содержит другой мы говорим про связь Has-a. Предназначение этой связи может быть разное, один объект может просто ссылаться а может и делегировать ряд задач какому-то другому объекту.

Давайте представим что у нас есть система задач. Пользователь может создавать списки задачи, а внутри списков размещать сами задачи. К каждой задаче можно прикрепить текст задачи и пользователя который ей занят.

Мы можем очень эффектно выразить это с помощью следующих классов:

- User
- List
- Task

Вот как они выглядят

Фрагмент 1.43

```
class User
{
    public string Name;
```

```
    public User(string name)
    {
        Name = name;
    }
}

class List
{
    public Task[] Tasks;

    public List(Task[] tasks)
    {
        Tasks = tasks;
    }
}

class Task
{
    public User Worker;
    public string Description;

    public Task(User worker, string
description)
    {
        Worker = worker;
        Description = description;
    }
}
```

Это не один возможный способ организации подобного примера. Текущая композиция например

позволяет задаче быть на нескольких досках сразу. Мы можем это исправить убрав ссылку на задачу из листа и сделав ссылку на лист в задаче.

С организацией и дизайном кода можно играться очень долго исходя из задачи.

Инкапсуляция

Каждый объект в системе имеет определенный контракт взаимодействия. Контракт выражается в операциях, которые можно выполнить над объектом:

- Что нужно предоставить ему для корректной работы;
- Что в состоянии объекта может изменяться в процессе работы;
- Что он нам готов предоставить в результате.

Одна из задач проектирования кода – это построение прозрачных контрактов и абстракций. Если контракт понятен, лёгок в использовании и не привносит неожиданностей, то такой контракт хороший.

Абстракция – это, в каком-то смысле, и есть контракт. Мы абстрагируемся от несущественных деталей реализации, и нас также ограничивают от неправильных действий над объектом. Абстракция также накладывает определенный контракт на реализацию. Но в некоторых ситуациях эти слова не взаимозаменяемые: так, при реализации контракта, мы будем писать конкретные строки кода, которые и будут его выражать. Назвать абстракцией этот код не получится.

Описывая класс, мы уже описываем формальный контракт о том, что объект этого типа будет иметь определённые члены. Но также в классы мы сразу записываем реализацию. В других главах этого курса мы познакомимся с построением абстракций без или с частичной реализацией.

В C# уже на этапе проектирования типа мы можем воспользоваться рядом средств, чтобы описать контракт будущего объекта:

1. Поля
2. Функциональные члены
3. Конструкторы

Благодаря этому мы можем указать данные, над которыми мы властвуем, закрыть их от прямого вмешательства. Сделать простые и понятные методы, которые будут по запросу работать с этими данными, и определить конструкторы, которые не позволяют с самого начала существовать в системе объекту с некорректным состоянием.

Состояние – это совокупность значений всех полей объекта.

Если какое-то значение меняется, то меняется и состояние объекта. Наша задача построить объект так, чтобы его состояние не становилось некорректным. Некорректное состояние – это

состояние, работа с которым сопряжена с ошибками и багами.

Первое, что нам нужно сделать, это определить: какую ответственность на себя берёт данный тип?

В нашем случае у нас есть задача:

В нашей игре есть автомат. Автомат заряжен патронами разных типов:

1. Бронебойный
2. Трасирующий
3. Обычный

Пока патроны отличаются лишь текстом, который выводится при стрельбе. У автомата есть очередь таких патронов, автомат также можно перезарядить. Перезарядка – это либо установка новой очереди, либо восстановление предыдущей.

Сейчас наша задача построить такой тип, который делал ровно то, что мы хотим, и нельзя было бы сделать того, что мы не ожидаем. На первый взгляд простая задача, не так ли?

Давайте возьмём такое решение:

```
class Gun
    {
        public int CurrentBullet;
```

```
        public List<string> Bullets;

        public void Shot()
        {
            if (CurrentBullet >=
Bullets.Count)
                return;


Console.WriteLine(Bullets[CurrentBullet]);
            CurrentBullet++;
        }

        public void Reload(List<string>
bullets = null)
        {
            if (bullets == null)
                Bullets = bullets;

            CurrentBullet = 0;
        }
    }
```

Как вы думаете, какие операции мы можем совершать над объектом данного типа?

Даже, на первый взгляд, можно сказать о следующих операциях:

1. Вызывать метод Shot (фактически стрелять этим оружием)

2. Вызывать метод Reload (фактически перезаряжать оружие)

Это мы поняли, взглянув на публичные методы оружия. Но ограничивается ли спектр наших операций этим?

Нет, ещё мы можем делать такие непреднамеренные операции:

1. Произвольное изменение текущего патрона в очереди. Ожидает ли данный тип, что снаружи это значение можно изменить как угодно?
2. Возможно заменить обойму без сбрасывания указателя на текущий патрон.

И это не всё. Пока что я придержу все карты. Согласитесь, что если мы загрузим обойму через Reload на 10 выстрелов, а потом сделаем 5 выстрелов, то ничего страшного не произойдёт. Но что будет, если мы самостоятельно, без Reload, поставим в поле Bullets обойму, скажем, на 3 патрона, и попробуем сделать выстрел? Внезапно оружие не будет стрелять. А что, если перед этим мы произвели всего два выстрела? Оно выстрелит, но один раз. Не совсем очевидное поведение для того, кто этот код использует не правда ли? Но мы же сами разрешили менять это поле пользователю этим типом, соответственно нет ничего странного в том, что люди пробуют это делать.

Как мы можем защитится? Для начала нам нужно сделать поля private. Это позволяет закрыть доступ к ним вне типа. Добились ли мы защиты внутреннего состояния? Оно стало лучше, но всё ещё есть кое-какие проблемы. Они не критичны в данном типе, но могут вызывать проблемы в других ситуациях.

Обратите внимание на метод Reload. Что он принимает? List<T>. А этот тип ссылочный. Это значит, если нам дадут очередь для стрельбы, то ссылку на эту очередь будет иметь тот, кто нам её дал. И, конечно же, он сможет её произвольно изменять, как ему это хочется. Если бы у нас был чувствительный код к этому, то мы бы получили множество багов. Что нужно делать в данном случае? Ограничивать абстракцию. Конечно же, нам хватило бы тут и обычного массива. List<T> добавляет операции записи и удаления, и мы, по сути, сами сказали: “дайте нам ссылку на что-то, что без нашего ведома может изменяться, а именно пополняться новыми элементами или стремительно сокращаться”.

Если подвести итог, то инкапсуляция – это защита внутреннего состояния объекта от непреднамеренного воздействия. Эта защита может достигаться множеством путей: правильной декларацией типов, использованием модификаторов доступов и многим другим. В нашем случае, мы рассмотрели первые два.

Наследование (Is-a)

Когда мы работаем с классам и объектами мы можем связывать их не только с помощью ссылок между объектами (Has-a) но наследуя классы друг от друга (Is-a). Это удобно когда мы имеем несколько классов у которых есть нечто общее. При этом собрать все классы в один мы не можем.

Наследование нам также нужно для реализации полиморфизма подтипов. Обычно чисто наследование, без техник полиморфизма мы не используем. Но я постараюсь привести пару примеров.

Фрагмент 1.44

```
class Knight
{
    public int Health;
    public int Armor;

    public void TakeDamage(int damage)
    {
        Health -= damage - Armor;
    }

    public void Pray()
    {
        Armor += 1;
```

```
    }
}

class Barbarian
{
    public int Health;
    public int Armor;

    public void TakeDamage(int damage)
    {
        Health -= damage - Armor;
    }

    public void Waaaagh()
    {
        Health += 10;
        Armor -= 1;
    }
}
```

Как вы видите между этими классами много общего. Мы могли бы избавиться от дублирующегося кода с помощью связи Has-a, сделав третий класс Health и сделав ссылку у варвара и рыцаря на какое-то здоровье. И это могло бы сработать и чаще всего так и делается. Но не всегда. Иногда лучше сделать связь is-a, сказав что и варвар и рыцарь является чем-то, что содержит здоровье.

Фрагмент 1.45

```
class Warrior
{
    public int Health;
    public int Armor;

    public void TakeDamage(int damage)
    {
        Health -= damage - Armor;
    }
}

class Knight : Warrior
{
    public void Pray()
    {
        Armor += 1;
    }
}

class Barbarian : Warrior
{
    public void Waaaagh()
    {
        Health += 10;
        Armor -= 1;
    }
}
```

В данном случае мы сделали новый класс “воин”. И сказали что и рыцарь и варвар является воином а

значит они содержат всё тоже, что содержит класс воин. Мы как бы перенесли всё, что внутри класса воин в классы рыцарь и варвар. При этом и рыцарь и варвар обладают уникальным возможностями. Рыцарь может помолится а варвар сделать Ваааrх.

Цепочки вызовов конструкторов

Когда базовый класс (тот от когда наследуемся) имеет конструктор, то мы должны его вызвать в производном классе (тот который наследуется). Сделать мы можем это с помощь цепочки вызова конструкторов. При этом конструктор базового класса вызывается первым.

Фрагмент 1.46

```
class Warrior
{
    public int Health;
    public int Armor;

    public Warrior(int health, int armor)
    {
        Health = health;
        Armor = armor;
    }

    public void TakeDamage(int damage)
    {
```

```
        Health -= damage - Armor;
    }
}

class Knight : Warrior
{
    public Knight(int health, int armor) :
base(health, armor) { }

    public void Pray()
    {
        Armor += 1;
    }
}

class Barbarian : Warrior
{
    public int LenghtOfAxe;

    public Barbarian(int health, int armor,
int lenghtOfAxe) : base(health, armor)
    {
        LenghtOfAxe = lenghtOfAxe;
    }

    public void Waaaagh()
    {
        Health += 10;
        Armor -= 1;
    }
}
```

В базовом классе (воин) мы добавили конструктор для инициализации здоровья и брони. Это привело к тому, что нам пришлось определять конструкторы в производных классах со сходными параметрами для вызова конструктора базового класса.

Это происходит благодаря двоеточию после имени конструктора и ключевому слова base. Это и есть вызов конструктора базового класса. Мы можем передать ему те же значения что передали нам, а может подставить какие-нибудь литералы.

Фрагмент 1.47

```
public Barbarian(int armor, int lenghtOfAxe)
: base(100, armor)
{
    LenghtOfAxe = lenghtOfAxe;
}
```

Например в этом случае все варвары создавались бы с фиксированным здоровьем и мы могли бы назначить им только броню.

Модификатор доступа protected

Базовый класс – это отдельный класс. Как бы это не звучало очевидно. Но вам стоит держать мысль о том, что раз класс отдельный то и проектируется он отдельно. Т.е нужно жестко сливать два класса.

Базовый класс может содержать члены которые доступны производным классам. А может и сокрывать члены от всех, в том числе и от базовых классов.

При использование модификатора private член не доступен производному классу. А при использование protected член доступен только самому классу (как в случае с private) и в производном классе.

Абстракция

Смысловая абстракция

Фактически, мы имеем бесконечное число связей и свойств объекта, воссоздание которых не имеет, во-первых смысла, а во-вторых фактически не возможно.

В такие моменты к нам приходит на помощь абстракция. Абстракция – это отвлечение от несущественных деталей объекта в пользу более существенных.

Представим, что у нас есть кот. Как описать кота в программе?

Я могу придумать следующие отличительные свойства кота:

- У него есть длина усов
- Длина хвоста
- Длина от хвоста до усов
- Цвет шерсти

И вот я создаю класс в программе:

Фрагмент 1.48

```
class Cat
```

```
{
    public int WhiskerLength;
    public int TailLength;
    public int Length;
    public Color Color;
}
```

Сижу, пержу, смотрю вдаль и начинаю понимать, что суть программы не в моделировании общего кота, а в функционале. А кот – это лишь смысловая единица, через которую я описываю функционал моей программы.

И что, на самом деле, функционал программы заключается в том, что когда я на кота кликаю, они издаёт один из 4 заранее записанных звуков. И мы такие опа...

Оказывается, что кот – это абстракция от функционала генерации случайного звука из списка записанных, а также обработчик нажатия. И назвали мы эту сущность – кот. В итоге для этого нам понадобится, конечно же, другой класс.

В данном случае мы произвели абстракцию от того, что фактически делает код в сторону понятной сущности. Почему именно кот? Потому, что у нас на экране показывается кот, а приложение называется SweetCat. Это название логичное и очевидное.

Могли бы мы назвать получившийся класс как “RandomSoundOnClick”? Да, почему нет. Но таким образом мы бы пересказали внутренне устройство, и это было бы слишком буквально. Такая буквальность имеет смысл уровнем ниже. Но когда мы поднимемся выше, например, в код, который задаёт начальные данные на уровень, то там хотелось бы видеть нечто подобное:

Фрагмент 1.49

```
var cat = new Cat(listOfSounds, Color.Gray);
EventSystem.RegistClickHandler(cat);
```

Всем сразу понятно, что тут конфигурируют кота.

Грубо говоря то, что сейчас сделано – это абстракция для программиста. Абстракция от тонкостей реализации для понимания предназначения и глобального замысла в рамках всей программы и решаемой задачи.

Но также у нас есть части нашей системы, которые имеют свой взгляд, которому важны другие детали. Например, у нас уже появилась система обработки событий, которая принимает некий объект и регистрирует его в системе. Когда происходит событие, то объект об этом оповещается. Система используется многими модулями программы. Например, не только кот хочет знать, что произошло

событие, но и условная картина в игровом пространстве, которая реагирует на клик по-своему.

Фрагмент 1.50

```
var picture = new Picture(image);
EventSystem.RegistClickHandler(picture);
```

Неужели системе событий придётся с каждым случаем работать уникально? По-своему с картинами и по-своему с котом? На самом деле нет. Мы можем создать другую абстракцию.

Системе обработки событий нет никакого дела до того картина это или кот. Об этом, кстати, говорит один из принципов проектирования, ISP – принцип разделения интерфейсов, суть которого в том, что сущности при работе с другими сущностями должны знать необходимый минимум. Для системы событий, конечно же, излишне знать о цвете кота.

Для такой системы нужен свой взгляд на вещи. В нашем случае через абстракцию “ОбработчикСобытияКлика”. И мы будем говорить, что системе событий нужен некий обработчик события клика, и что у нас есть класс кот и книга, которые являются обработчиками события клика.

Звучит просто? Это и является довольно простым концептом, который открывает много возможностей

и опасностей. Мы уже поняли, как это выражать на русском языке, и, как вы смогли догадаться, тоже самое мы можем выразить в программе на языке C#. И дальше будет именно это.

Формальная система типов и абстракция

И так представим, что у нас есть некое абстрактное оружие. У оружия есть интерфейс взаимодействия – стрелять. Некая функция-глагол.

Когда мы говорим оружию у нас в руках стрелять, оно стреляет. Но есть разные реализации оружия: дробовик – осыпает врагов дробью, а пистолет делает точные и слабые выстрелы, и при этом у него не кончаются патроны.

В таком случае у нас есть абстракция – оружие. Эту абстракцию ещё можно назвать общим случаем. И есть реализации, более частные случаи: дробовик и пистолет.

Само по себе обобщение не всегда нужно, но даже при описании такой игры мы будем прибегать к описанию взаимодействия с абстрактными оружием, не повторяя одно и тоже для каждого частного случая. Но такое обобщение необходимо в таких строгих системах типов, как в C#.
Давайте представим пример, что игроку в руку попало не оружие, а предмет другого типа, например, противник с мечом. Будет ли такая операция ошибочной в нашей системе? А что будет, если игрок попробует стрельнуть таким “оружием”? Будет баг. Сложно сказать, что произойдёт, но система выйдет

из планируемого поведения, и посыпаться может что угодно.

Соответственно, это было бы возможно, если бы у нас не было формально закреплённой абстракции, и наш язык позволял присваивать в некое поле “ТекущееОружие” объект любого типа. Но так нам делать нельзя.

Но с другой стороны, слишком строгий контроль не позволял бы нам делать более общим “ТекущееОружие”, и присваивать туда то объект типа “Пистолет”, то типа “Дробовик”.

Решение и становится полиморфизмом подтипов. Когда некий тип A формирует контракт и либо частично его реализует (в т.ч. и не реализует вообще), либо позволяет переопределить реализацию, и есть некий тип B, который реализует не реализованное, либо переопределяет уже реализованное.

В нашем примере, тип A – это некий тип “Оружие”, который определяет общий интерфейс взаимодействия со всем оружием. А тип B – это, например, тип “Пистолет”, который реализует этот интерфейс, т.е. какие-то конкретные действия при абстрактном “выстреле”.

Виртуальные методы

Один интерфейс – множество реализаций

Бьерн Страуструп

Для работы с формальными абстракциями в C# предусмотрено множество методов. Даже модификаторы доступа, так или иначе, помогают нам возводить абстракцию, но сейчас нас интересуют основные синтаксические методы для работы с полиморфизмом подтипов.

Когда мы наследуем некий тип “B” от типа “A”, мы можем не только расширить тип “A”, но и переопределить часть его поведения. В такой ситуации мы и получаем: тип “A” задаёт некий интерфейс, а мы можем, соблюдая его, переопределить поведение.

Для этого базовый класс должен указать, что некий его метод виртуальный, и тогда производный класс может переопределить поведение этого метода.

Хорошим тоном является не переопределять поведение базового класса, а лишь дополнять его. Про это говорит, например, один из принципов SOLID – LSP.

Фрагмент 1.51

```
class Program
{
    static void Main(string[] args)
    {
        Gun gun = new Gun();
        Player player1 = new Player();
        gun.Fire(player1);
    }
}

class Player
{
    private float _health;

    public void ApplyDamage(float amount)
    {
        _health -= amount;
    }
}

class Gun
{
    private int _bullets;
    private float _damage;

    public void Fire(Player player)
    {
        if (_bullets <= 0)
            return;

        player.ApplyDamage(_damage);
```

```
        _bullets--;
    }
}
```

У нас есть некое оружие, которое стреляет в игрока. Оружие наносит игроку урон и отсчитывает патроны. Также у нас есть правило – оружие без патронов не стреляет.

Всё использование оружия заключено в методе Main. Но оружие может пользоваться, и какой-то тип, который может симулировать бой. Например:

Фрагмент 1.52

```
class Battle
{
    private Gun _gun;
    private Player[] _players;

    public Battle(Gun gun, Player[] players)
    {
        _gun = gun;
        _players = players;
    }

    public void Simulate()
    {
        foreach (var player in _players)
        {
            _gun.Fire(player);
```

```
        }
    }
}
```

И тут мы захотели иметь разные реализации оружия:

- Бесконечный пистолет;
- Лук, у которого с каждым выстрелом уменьшается урон.

Как мы можем это выразить? На самом деле просто. И, я думаю, вы бы с лёгкостью с этим справились. Но я добавлю ещё одно условие: нужно сделать такое оружие, и при этом, чтобы алгоритм симуляции боя и какие-нибудь другие части программы не изменялись.

Т.е. хотелось бы работать с этим извне, на уровне конфигурации, и не переписывать код каждый раз, когда нам добавляется оружие.

Мы можем удовлетворить условие с помощью виртуальных методов. Мы можем пометить метод Fire в базовый класс модификатором virtual, а в производных классах сделать точно такие же методы по сигнатуре, но с модификатором override.

Это не имеет смысла, если вы не узнаете, что ссылка базового типа может ссылаться на производный тип.

И при этом, если по такой ссылке вызывается виртуальный метод, то фактически будет вызван метод производного типа, если он там переопределён.

Фрагмент 1.53

```
class Program
{
    static void Main(string[] args)
    {
        Gun gun = new Gun();
        Player player1 = new Player();
        gun.Fire(player1);

        gun = new Bow();
        gun.Fire(player1);
        gun.Fire(player1);

        Battle battle = new Battle(new
Pistol(), new Player[] { player1 });
        battle.Simulate();
    }
}

class Battle
{
    private Gun _gun;
    private Player[] _players;

    public Battle(Gun gun, Player[] players)
```

```
    {
        _gun = gun;
        _players = players;
    }

    public void Simulate()
    {
        foreach (var player in _players)
        {
            _gun.Fire(player);
        }
    }
}

class Player
{
    private float _health;

    public void ApplyDamage(float amount)
    {
        _health -= amount;
    }
}

class Gun
{
    private int _bullets;
    protected float Damage;

    public virtual void Fire(Player player)
    {
```

```
        if (_bullets <= 0)
            return;

        player.ApplyDamage(Damage);
        _bullets--;
    }
}

class Pistol : Gun
{
    public override void Fire(Player player)
    {
        player.ApplyDamage(Damage);
    }
}

class Bow : Gun
{
    public override void Fire(Player player)
    {
        base.Fire(player);
        Damage /= 2;
    }
}
```

Обратите внимание, что в этом коде добавилось два класса: Bow и Pistol, в которых произошло то, о чём мы говорили. В базовом классе Gun метод Fire помечен как virtual, а в производных Bow и Gun присутствуют эти же методы, но уже как override.

Потом посмотрите на метод Main. Обратите внимание, как в переменную с типом Gun мы кладём Bow, а потом при создании объекта типа Battle в конструктор, который требует тип Gun, я передаю тип Pistol. Синтаксически, когда мы указываем тип Gun, мы можем использовать только то, что определено в этом типе, но при этом, то, что там определено, может быть переопределено в производных типах. Поэтому иногда мы работаем, вроде бы, с Gun, но фактически это Pistol. Благодаря этому, мы имеем некоторую степень защищённости и, при этом гибкости.

Из важных мелочей:

- Damage стал protected для того, чтобы производные классы могли его изменить;
- В классе Bow мы используем строку, которая вызывает метод базового класса. В случае с Pistol мы полностью переопределяем код. Т.е. метод Fire базового класса не выполнится. А вот в случае с Bow мы вызываем метод базового класса и добавляем ему дополнительное поведение (уменьшение урона в два раза).

Задача с собеседования

На собеседовании вам могут задать интересный вопрос, который на практике встречается не часто, из соображений дизайна кода, но понимать это

безусловно полезно. Суть вопроса заключается в том, как происходит переопределение методов на нескольких уровнях иерархии.

```
class A
{
    public virtual Do() =>
Console.WriteLine("A");
}
class B : A
{
    public override Do() =>
Console.WriteLine("B");
}
class C : B
{
    public override Do() =>
Console.WriteLine("C");
}

class Example
{
   public static void Main(string[] args)
   {
       A a = new C();
       a.Do(); //1

       a = new B();
       a.Do(); //2
   }
}
```

Что в таком случае будет выводиться в консоль? Ответ очевиден:

1. C
2. B

Наследники переопределяют метод, при этом вызывается тот, что будет ниже по иерархии. Наверное, не стоит говорить, что вызывается метод фактического типа, а не просто самого нижнего наследника.

То есть, если у нас есть ссылка типа “A” на объект типа “C”, то при вызове метода по типу “A” будет происходить поиск переопределённого метода сверху вниз до самого нижнего, а так как там тип “C”, то будет вызываться его метод. Если вместо “C” будет “B”, то будет вызываться метод “B”.

Учитывая это, что будет в таком случае?

```
class A
{
    public virtual Do() =>
Console.WriteLine("A");
}
class B : A
{
    public virtual Do() =>
Console.WriteLine("B");
}
class C : B
```

```
{
    public override Do() =>
Console.WriteLine("C");
}

class Example
{
   public static void Main(string[] args)
   {
       A a = new C();
       a.Do(); //1

       a = new B();
       a.Do(); //2
   }
}
```

Ответ может запутать:

1. A
2. A

Дело в том, что “B” заменяет метод на свой, и делает его виртуальным, “B” имеет по сути новый метод, а “C” его переопределяет, и к виртуальном методу типа “A” это уже не имеет значения.

С точки зрения стилистики, чтобы было меньше путаницы, мы должны пометить метод в типе “B” модификатором new.

```
class B : A
{
    public new virtual Do() =>
Console.WriteLine("B");
}
```

Интерфейсы и абстрактные методы

Часто нам не нужна никакая реализация. В случае с оружием, это не так. Нам очевидно, что всё оружие имеет боезапас (кроме пистолета), и наносит урон. Но иногда у нас абстракция не имеет никакой реализации (интерфейс) или имеет частичную (абстрактный класс с абстрактными методами). В обоих случаях нам при наследовании этих типов нужно их полностью реализовать.

Начнём с интерфейсов. Сразу стоит оговорить, что интерфейсы наследуются так же, как и обычные классы, но интерфейсов мы можем унаследовать сколько хотим, в отличие от обычных классов. Но интерфейсы также формируют тип, и с этим типом мы можем определять переменные, параметры, поля и многое другое.

Давайте представим, что у нашего игрока есть некоторый инвентарь. В нём могут быть размещены самые разные предметы, и мы можем с ними взаимодействовать. По сути взаимодействие с предметом сводится к тому, что:

- Мы помещаем предмет в список предметов в инвентаре.
- При помещении предмета в инвентарь, мы оповещаем его об этом.

- При выбросе из инвентаря оповещаем об этом.

Решить эту задачу мы можем так:

Фрагмент 1.54

```
class Gun : IInventoryHandler
{
    private int _bullets;
    protected float Damage;

    public virtual void Fire(Player player)
    {
        if (_bullets <= 0)
            return;

        player.ApplyDamage(Damage);
        _bullets--;
    }

    public void OnPickup()
    {
        //off scene model
    }

    public void OnDrop()
    {
        //on scene model
    }
}
```

```
interface IInventoryHandler
{
    void OnPickup();
    void OnDrop();
}
```

Мы просто создали тип IInventoryHandler, через ключевое слово interface. Такой тип может содержать часть функциональных членов: свойства, методы и индексаторы. Но при этом, эти члены не содержат реализации, они как бы пустые.

В дальнейшем класс или структура может наследовать этот интерфейс и реализовывать его. При наследовании интерфейса его необходимо полностью реализовать. Так, в классе Gun появились методы OnPickup и OnDrop, которые будут вызываться при подборе\выбросе этого предмета. У нас считается, что оружие 3D модель, и при вызове этих методов оружие включает\выключает свою модель.

Вероятней всего, у нас все предметы будут с 3D моделью, но предположим, что пока только оружие. Что до остальных предметов, поведение при подборе\выбросе другое.

Пример работы:

Фрагмент 1.55

```
class Program
{
    static void Main(string[] args)
    {
        Gun gun = new Bow();
        Player player = new Player();

        player.PickUp(gun);
    }
}

class Player
{
    private float _health;
    private List<IInventoryHandler>
_inventory = new List<IInventoryHandler>();

    public void ApplyDamage(float amount)
    {
        _health -= amount;
    }

    public void PickUp(IInventoryHandler
item)
    {
        _inventory.Add(item);
        item.OnPickup();
    }
}
```

В случае с абстрактными классами, мы имеем обычный класс, но из которого, как и из интерфейса, мы не можем создать объект. Абстрактный класс мы можем унаследовать, но при этом мы должны реализовать все его абстрактные члены.

С помощью абстрактных классов удобно возводить каркасы для наследников. Внутри такого класса мы можем, например, описать порядок действий, и даже реализовать некоторые шаги, а наследников просим, как бы, подставить код в определённые точки.
Так мы закрепим шаги алгоритма в абстрактном классе, а в производных уже будем выполнять конкретные действия.

Давайте разберём пример, в котором у нас есть башни, которые ждут противника в зоне поражения, выбирают его как цель и начинают атаку. При этом наследники занимаются следующим:

- Определяют можно ли атаковать эту цель;
- Атакуют эту цель.

Фрагмент 1.56

```
abstract class Tower
{
    private Player _target;

    public void Update()
    {
```

```
        if(_target == null)
        {
            foreach (var player in
GetClosestPlayers())
            {
                if(CanAttack(player))
                {
                    _target = player;
                    break;
                }
            }
        }

        if (_target != null)
            Attack(_target);
    }

    protected abstract bool CanAttack(Player
player);

    protected abstract void Attack(Player
player);

    private IEnumerable<Player>
GetClosestPlayers()
    {
        throw new NotImplementedException();
    }
}

class ArcherTower : Tower
```

```
{
    protected override void Attack(Player
player)
    {
        player.ApplyDamage(2);
    }

    protected override bool CanAttack(Player
player)
    {
        throw new NotImplementedException();
    }
}
```

Как вы видите, мы определили логику производных классов как абстрактные методы, а внутри производного класса ArcherTower реализовали абстрактные методы с помощью override. Часть методов у нас не имеют реализацию только, чтобы показать общий пример.

Эту абстракцию можно переделать по-разному. Например, мы понимаем, что почти все башни имеют задержку между выстрелами, и эта логика будет дублироваться, поэтому нам понадобится промежуточный класс. Также не все башни атакуют, некоторые замедляют цели, а некоторые работают только по дружественным целям и лечат их.

Вендинговый автомат

Описание проекта

Как часто вы путешествуете? Мне приходится иногда отправляться в путь на очередную конференцию или выступление и мне очень нравится приезжать на вокзал или в аэропорт за 20-30 минут до отбытия. Обычно я нахожу уютное место, включаю музыку и представляю, что будет происходить сразу, как только я десантируюсь в месте моего назначения.

Этот ритуал, несомненно, проходит под шипение газировки и хруст упаковки с бисквитными пирожными. Их я беру из удобных автоматов, в которые я могу загрузить наличку и получить мои маленькие радости, но больше всего я люблю оплачивать покупки картой.

К сожалению, наш мир сложней, нежели компьютерная программа, и выдача продукта заключается не только в переключение флага “IsOrderComplete”. Автомату необходимо произвести некоторые механические действия для того, чтобы его содержимое было доставлено пользователю в руки или хотя бы в отделение, до которого может дотянуться покупатель.

Бывают ситуации, когда аппарат сделал все необходимые действия, но по той или иной причине товар не был доставлен. Например, при горизонтальном падении он может встать под углом.

В таком случае автомату нужен надежный механизм для контроля наличия выдачи. В разных ситуациях это может происходить по-разному и каждое решение обладает как преимуществами, так и недостатками. А также, в некоторых ситуациях можно выбрать более дешевое средство, если контекст сглаживает недостатки.
Представьте, что вы тот разработчик, который пишет программный код для этой машины. Ваше техническое задание написано на той же салфетке, на которой вам написали сумму вознаграждения за выполненный проект.

Выглядит оно примерно так:

Из неё мы понимаем,что:

1. Аппарат должен уметь принимать карты и монеты. Иногда кардридер может быть не подключён, аппарат должен не паниковать при этом;
2. Есть разные контроллеры доставки. Их будут разрабатывать другие разработчики. Но их будет много и они будут меняться;
3. Покупатель может забрать сдачу;
4. Есть разные хранилища, которые доставляют товар до отдела выдачи. Они также разрабатываются другими разработчиками;

Представили всё это? А теперь давайте полностью погрузимся в нашего вымышленного аватара и попробуем решить эту задачу.

Массивы и циклы

Ваша задача

Сейчас вам необходимо написать немного кода. В связи с тем, что мы идем последовательно, сейчас вам нельзя объявлять свои функции, использовать классы и объекты, и много чего ещё. По сути, в вашем распоряжении только массивы и циклы. Ну а также условные операторы, арифметические операторы и по мелочи.

Такое ограничение поставлено не просто так. Я напоминаю, что это эволюционные примеры, которые сначала решают задачу самыми примитивными способами, с которыми знаком начинающий разработчик, а только потом последовательно вводят новые темы, показывая зачем оно надо.

Из-за того, что мы разрабатываем консольное приложение, у нас меньше пространства для творчества. Но при этом всё сильно упрощается. Мы не программируем настоящий автомат, а мы скорее притворяемся, что у нас есть настоящий автомат. Эта условность может поначалу смутить, так как некоторые вещи мы будем представлять по-разному. Но не стоит паниковать.

Когда мы обсуждаем что-то, мы концентрируемся только на том что обсуждаем, абстрагируясь от ненужного нам. Это позволяет нам разбирать различные темы не вдаваясь в ненужные детали строения настоящего автомата. В программировании мы вообще очень часто притворяемся и делаем вид, что что-то является чем-то.

Чтобы вы понимали, что от вас требуется, давайте я вам задам жесткие рамки с точки зрения использования того, что у вас получится. Сделаю я это с помощью команд, которые можно отправлять вашему автомату и описанных реакций на них. Вы поймете, что вы закончили сразу, как только ваш автомат сможет выполнять перечисленное ниже.

Под отправкой команды я подразумеваю ввод строк в консоли:

AddMoney
Добавляет на баланс автомата деньги;

GetChange
Автомат обнуляет баланс и выдаёт сдачу доступными монетами, номиналом: 1, 2, 5 и 10 рублей;

BuyGood {id} {count}
Автомат выдает товар в определенном количестве и снимает деньги с баланса;

При старте автомат сам загружается монетками, товарами и выбирается какой-то из модулей оплаты. Всего есть два модуля: карта и наличка.

1. Если оплата картой – то баланс зачисляется, при этом количество доступных монет не изменяется:
2. Если оплата наличкой, то при добавлении баланса нужно указать какими монетами происходит оплата и они добавляются в список доступных монет.

Помните модуль проверки выдачи? Та штука, которая проверяет, что вы действительно получили заказ? А помните, я говорил, что мы часто притворяемся? Так вот, вам так же нужно притворится, что у вас есть такой модуль, и, что примерно 3% заказов не выдаются. Если заказ не был выдан, то деньги с баланса пользователя не снимаются.

Массивы и циклы.Возможное решение

Давайте начнём писать максимально примитивный код. Работать мы будем в обычном консольном проекте. Потому, вам достаточно открыть Visual Studio и создать консольное приложение. Если у вас его нет, то вам стоит в установщике указать пакет “Разработка классических приложений .Net” или просто найти этот проект в репозитории.

Сейчас мы будем работать в рамках класса *programm* в методе *Main*. В последующих этапах мы, конечно же, проведём функциональную декомпозицию. Полный листинг этого этапа вы можете найти в *WendingMachine\ArraysAndCycles\Programm.cs*

Начнём с описания состояния программы. В начале метода Main я определил 6 переменных, первая нужна для хранения текущего баланса. Вторая и третья для доступных монет. В массивах определено то, сколько монет, какого номинала осталось и, соответственно, номинал этих монет. Четвёртая служит для выбора текущей системы оплаты. А три остальных вкупе образуют данные товара.

Фрагмент 2.1

```
int balance = 0;
int[] coinsQuantity = { 0, 0, 0, 0}; //1, 2,
5, 10
```

```
int[] coinsValues  = { 1, 2, 5, 10};
PaymentType payment = PaymentType.Card;
string[] names = { "Шоколадка", "Газировка"
};
int[] prices = { 70, 60 };
int[] availableQuantity = { 5, 2 };
```

Да, эти три массива жутко неудобны. Работает это таким образом: определенный индекс подходит ко всем трем массивам. Применяемый к определенному массиву, он позволяет узнать те или иные характеристики товара. Здесь представлены массивы для трёх характеристик: имя, цена и остаток.

В такой организации есть проблема – мы можем добавить имя и забыть добавить цену. Тогда, в дальнейшем программа не будет работать корректно. Т.е тут отсутствует целостность данных. Но эй, вы же помните, что мы крутимся, как можем? Но проблему мы уже видим и в будущем совершим героический подвиг – избавимся от неё.

Идём дальше, теперь нам нужно организовать цикл обработки команд в приложении с консолью. Он заключается в том, чтобы запустить бесконечный цикл, каждая итерация которого, считывает с консоли строку и интерпретирует ее с помощью операторов управления. В дальнейшем, мы попробуем написать гибкую систему через АОП, а

пока что, куча условных операторов. Выглядит это так:

Фрагмент 2.2

```
string command = "";
while (true)
{
    Console.Clear();
    Console.WriteLine($"Баланс {balance}");
    Console.WriteLine("Введите команду:");
    command = Console.ReadLine();

    if(command == "AddMoney")
    {

    }
    else if(command == "GetChange")
    {

    }
    else if (command.StartsWith("BuyGood"))
    {

    }
    else
    {
        Console.WriteLine("Команда не
определена");
    }
```

```
    Console.ReadLine();
}
```

Как вы видите, мы и впрямь гоняем бесконечный цикл и каждый раз, когда мы оказываемся внутри него, мы сообщаем пользователю баланс (с помощью интерполяции строк) и просим ввести строку, которую мы попытаемся распознать как команду.

Сразу после, мы напрямую проверяем строку, и если она равна строке например “AddMoney”, то выполняем команду, которая скрыта под этим именем. В случае с AddMoney — добавление баланса.

Что теперь? Нужно сделать имплементацию команд.

Реализация AddMoney

Начнём с добавления денег на баланс. Помните правило? В зависимости от модуля оплаты, начисление происходит по-разному.

Фрагмент 2.3

```
if(command == "AddMoney")
{
    switch (payment)
    {
        case PaymentType.Coins:
            for(int i = 0; i <
coinsValues.Length; i++)
            {
                Console.WriteLine($"Сколько
монет номиналом {coinsValues[i]} вы хотите
внести?");
                int count = 0;
                while
(!int.TryParse(Console.ReadLine(),
                                        out
count))
                {
                    Console.WriteLine("Вы
ввели не число!");
                }
                coinsQuantity[i] += count;
                balance += count *
coinsValues[i];
```

```
            }
            break;
        case PaymentType.Card:
            Console.WriteLine("Сколько снять с
вашей карты?");
            int balanceDelta = 0;
            while
(!int.TryParse(Console.ReadLine(),
                    out balanceDelta))
            {
                Console.WriteLine("Вы ввели
не число!");
            }
            balance += balanceDelta;
            Console.WriteLine("Баланс успешно
пополнен");
            break;
        default:
            break;
    }
}
```

После того, как мы определились в том, какую команду хочет выполнить пользователь, мы начинаем работать с внутренним состоянием автомата. Для начала мы проверяем то, какой способ оплаты стоит: монеты или карта.

Выбор происходит с помощью оператора switch.

Если выбраны монеты:

Мы по очереди предлагаем пользователю внести монеты доступных номиналов. Эти номиналы прописаны в массиве coinsValues. После ввода количества монет, определенного номинала, мы увеличиваем баланс автомата.

Если выбрана карта:

Мы просто спрашиваем сколько пользователь хочет внести средств и пополняем баланс на эту сумму через увеличение переменной balance.
*Давайте предположим, что пользователь не вводит отрицательные числа.

Здесь мы уже можем видеть дублирующийся код. В случае оплаты монетами нам нужно запрашивать у пользователя некоторое число до тех пор, пока он его не введёт. Это сопряжено с тем, что он может вводить не число или число выходящее за границу вместимости типа int.

Это делается через специфичную конструкцию на основе цикла while.

Фрагмент 2.4

```
while (!int.TryParse(Console.ReadLine(),
                          out count))
```

```
{
    Console.WriteLine("Вы ввели не число!");
}
```

Эта конструкция имеет явную тенденцию к дублированию. Дублирующийся код – это плохо. Ко мне часто во время моих лекций обращаются студенты, задающие вопрос: “Сколько раз должен продублироваться код, чтобы настало время ликвидировать это?”. Ответ всегда один: “Достаточно одного дублирования”. Подробней эту проблему и способы её решения мы обсудим дальше.

Реализация GetChange

Просто, лаконично и наповал убийственно.

Фрагмент 2.5

```
else if(command == "GetChange")
{
    balance = 0;
}
```

Если вам что-то тут непонятно – книгу лучше отложить.

Реализация BuyGood {id} {count}

Не так просто, не так лаконично и скорей всего заставит вас подняться и уйти на срочную чашку кофе.

Сложность реализации этой команды заключается в том, что нам нужно решить немного больше задач, чем в прошлый раз. Во-первых, нам нужно выполнить основную задачу, без которой все рядом стоящие аспекты не имеют смысла – продать товар. А именно снять деньги в зависимости от товара и его количества.

Во-вторых, нам нужно спарсить (примечание: парсинг – процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой.). В данном контексте под парсингом подразумевается разбор строки “10 5” на две переменные. id и count в типе int. После парсинга нам нужно произвести процесс валидации – проверки того, что данные корректные. Его не стоит смешивать с процессом парсинга, так как разбор множества чисел из некой последовательности символов не должен быть связан с моделью приложения (по крайней мере, в тех терминах, в которых работаем мы).

Здесь подразумевается валидация модели. Т.е если запрашивается товар под id 10, но его нет, то данные с одной стороны – корректные – id действительно представляется целым числом, как и просили. Но с другой стороны в данный момент такого товара нет.

Мы четко разделим реализацию команды на:

1. Разбиение строки на единицы данных;
2. Сопоставление этих данных с переменными (и их типами);
3. Проверка корректности этих данных на основе текущего состояния модели;

Перейдем к полной реализации:

Фрагмент 2.6

```
else if (command.StartsWith("BuyGood"))
{
    //Разбиение строки на единицы данных
    string[] rawData =
command.Substring("BuyGood ".Length).Split('
');

    //Сопоставление этих данных с переменными
(и их типами)
    if(rawData.Length != 2)
    {
        Console.WriteLine("Неправильные
```

```
аргументы команды");
        break;
    }

    int id = Convert.ToInt32(rawData[0]);
    int count = Convert.ToInt32(rawData[1]);

    //Проверка корректности этих данных
    //на основе текущего состояния модели.
    if(id < 0 || id >= names.Length)
    {
        Console.WriteLine("Такого товара
нет");
        break;
    }

    if(count < 0 || count >
availableQuantity[id])
    {
        Console.WriteLine("Нет такого
количества");
        break;
    }

    //Выполнение
    if(balance >= prices[id] * count)
    {
        balance -= prices[id] * count;
        availableQuantity[id] -= count;
    }
}
```

Тут вы можете наблюдать чёткие шаги, которые ведут нас к цели. Если действовать по принципу “разделяй и властвуй” – всё сложное можно привести к набору банальностей. В дальнейшем, мы эти банальности оформим более формально и всё станет ещё чётче.

Пока что нас с вами могут заинтересовать тонкости реализации. Всё начинается с небольшого отличия от других команд. Давайте вернёмся к общему каркасу (Фрагмент 2.2). Обратите внимание, что в операторах if и else if первые две команды сопоставляются через полное сравнение. А в случае с BuyGood мы используем метод StartWith.

Он позволяет проверить “начинается ли определённая строка с какой-то подстроки?”. Так мы проверяем, что если пользователь ввёл “BuyGood 10 5”, то всё хорошо, это наша команда. Проверка через равенство не сработала бы потому, что всё, что после BuyGood в строке, варьируется.

Далее начинается сама команда, с первого шага которой – мы достаем из строки команды наши данные, которые нужны для её выполнения (аргументы).

Фрагмент 2.7

```
string[] rawData = command.Substring("BuyGood
".Length).Split(' ');
```

Начинается это с использования метода Substring.

```
public string Substring(int startIndex)
```

Параметры

- startIndex
    - Type: System.Int32
    - Отсчитываемая от нуля позиция первого знака подстроки в данном экземпляре.

Возвращаемое значение:
Type: System.String
Строка, эквивалентная подстроке, которая начинается с startIndex в данном экземпляре или Empty, если значение startIndex равно длине данного экземпляра.

Т.е это просто функция, которая возвращает подстроку от строки. Мы её используем в таком контексте: просто вырезаем начало строки из строки, которую ввел пользователь. В результате мы получим только часть “10 5”, если пользователь введёт “BuyGood 10 5”.

Дальше, у этой строки мы сразу же вызываем метод Split. Он разбивает строку на массив строк по определённому признаку.

```
public string[] Split(params char[]
separator)
```

Параметры:

- separator
    - Type: System.Char[]
    - Массив символов, разделяющий подстроки в данной строке, пустой массив, не содержащий разделителей или null.

Возвращаемое значение:
Type: System.String[]

Массив, элементы которого содержат подстроки из этого экземпляра, разделенные символами из separator. Дополнительные сведения см. в разделе “Примечания”.

Разбор метода Split ищите в дополнительных материалах. Сейчас же мы с помощью него получили массив строк, содержащий два элемента {“10”, “5”}. И настает следующий шаг.

Фрагмент 2.8

```
if(rawData.Length != 2)
```

```
{
    Console.WriteLine("Неправильные аргументы
команды");
    break;
}

int id = Convert.ToInt32(rawData[0]);
int count = Convert.ToInt32(rawData[1]);
```

У нас есть массив какой-то длины. Сначала мы удостоверимся, что в нём есть нужное количество токенов, которое нас интересует. Делаем это через проверку длины массива. Если длины не хватает, то с помощью break прерываем цикл, что приводит пользователя к вводу новой команды.

Я думаю, вы заметили, что сначала мы пользователю сообщаем об ошибке, а потом прерываем выполнение. К сожалению, у нас такая структура программы, что это сообщение он не увидит, так как в начале цикла у нас происходит очистка консоли.

Но давайте перейдём к сути. Сразу после того, как мы убедились, что у нас есть необходимое количество данных. Мы начинаем превращать наши строки и распределять их по переменным с типом int.

Если пользователь введёт не число, конечно, у нас выбьется ошибка. И по хорошему нам нужно заменить вызов метода ToInt32 на int.TryParse.

Но видит бог я этого не хочу! Писать плохо – это искусство. Но я им сыт , я бы хотел в этот раз не показывать наглядный пример дублирования кода. Просто в следующем разделе создам необходимую нам функцию и просто воспользуюсь ей.

Фрагмент 2.9

```
if(id < 0 || id >= names.Length)
{
    Console.WriteLine("Такого товара нет");
    break;
}

if(count < 0 || count >
availableQuantity[id])
{
    Console.WriteLine("Нет такого
количества");
    break;
}

//Выполнение
if(balance >= prices[id] * count)
{
    balance -= prices[id] * count;
    availableQuantity[id] -= count;
```

```
}
```

Наконец-то мы добрались до фрагмента, в котором валидируем данные с точки зрения модели приложения и, если всё хорошо, совершаем продажу товара.

Все вместе эти команды и состояние нашего автомата формируют его. Давайте возьмем эту реализацию за отправную точку в нашем последующем прогрессе?

Листинг
WendingMachine/ArraysAndCycles/Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Code
{
    class Program
    {
        static void Main(string[] args)
        {
            int balance = 0;
            int[] coinsQuantity = { 0, 0, 0,
0}; //1, 2, 5, 10
```

```
            int[] coinsValues  = { 1, 2, 5,
10};
            string[] names = { "Шоколадка",
"Газировка" };
            int[] prices = { 70, 60 };
            int[] availableQuantity = { 5, 2
};
            PaymentType payment =
PaymentType.Card;

            string command = "";
            while (true)
            {
                Console.Clear();
                Console.WriteLine($"Баланс
{balance}");
                Console.WriteLine("Введите
команду:");
                command = Console.ReadLine();

                if(command == "AddMoney")
                {
                    switch (payment)
                    {
                        case
PaymentType.Coins:
                            for(int i = 0; i
< coinsValues.Length; i++)
                            {

Console.WriteLine($"Сколько монет номиналом
```

```
{coinsValues[i]} вы хотите внести?");
                                int count =
0;
                                while
(!int.TryParse(Console.ReadLine(),

out count))
                                {

Console.WriteLine("Вы ввели не число!");
                                }

coinsQuantity[i] += count;
                                balance +=
count * coinsValues[i];
                            }
                            break;
                        case
PaymentType.Card:

Console.WriteLine("Сколько снять с вашей
карты?");
                            int balanceDelta
= 0;
                            while
(!int.TryParse(Console.ReadLine(),
                                    out
balanceDelta))
                            {

Console.WriteLine("Вы ввели не число!");
```

```
                            }
                            balance +=
balanceDelta;

Console.WriteLine("Баланс успешно пополнен");
                            break;
                        default:
                            break;
                    }
                }
                else if(command ==
"GetChange")
                {
                    balance = 0;
                }
                else if
(command.StartsWith("BuyGood"))
                {
                    //Разбиение строки на
единицы данных
                    string[] rawData =
command.Substring("BuyGood ".Length).Split('
');

                    //Сопоставление этих
данных с переменными (и их типами)
                    if(rawData.Length != 2)
                    {

Console.WriteLine("Неправильные аргументы
команды");
```

```
                        break;
                    }

                    int id =
Convert.ToInt32(rawData[0]);
                    int count =
Convert.ToInt32(rawData[1]);

                    //Проверка корректности
этих данных на основе текущего состояния
модели.
                    if(id < 0 || id >=
names.Length)
                    {

Console.WriteLine("Такого товара нет");
                        break;
                    }

                    if(count < 0 || count >
availableQuantity[id])
                    {

Console.WriteLine("Нет такого количества");
                        break;
                    }

                    //Выполнение
                    if(balance >= prices[id]
* count)
                    {
```

```
                        balance -= prices[id]
* count;
                        availableQuantity[id]
-= count;
                    }
                }
                else
                {

Console.WriteLine("Команда не определена");
                }

                Console.ReadKey();
            }
        }
    }

    enum PaymentType
    {
        Coins,
        Card
    }
}
```

Функциональная композиция

Ваша задача

Продолжаем карнавал веселья и кода. Мы сделали работающую версию автомата. Но так ли она хороша? Я сейчас говорю не про техническую реализацию, а про стилистику кода и удобства работы с ним. Сейчас автомат представлен как монолитный блок кода, который наполнен дублированием, смешиванием ответственностей и неясностью происходящего.

Наша работа сейчас будет заключаться в функциональной декомпозиции. Мы попробуем разобрать нашу программу на набор функций, которые создадут нам слои абстракции и мы опишем в паре движений всю логику на верхнем уровне.

Ваша задача формируется очень просто, но слишком размыто.

Вам нужно:

1. Избавиться от дублирующегося кода;
2. Сделать программу понятней;
3. Сокрыть структуру данных и тонкости работы с ней;

Давайте разберёмся с каждым из пунктов по отдельности.

Избавиться от дублирующегося кода

Когда у вас есть два идентичных участка кода вам необходимо их ликвидировать. Как я писал выше, достаточно всего одного дублирования для решительных действий. В нашем коде полно таких моментов которые впоследствии испортят нам настроение и самое главное замедлят разработку.

В программу мы очень часто вносим изменения и наш код образует определённую логику. Каждый раз, когда эта логика меняется под влиянием изменений бизнес-процессов, нам приходится изменять N участков кода. В случае с небольшими блоками и при маленьких N – это сделать не так уж сложно.

Но основная проблема заключается в том, что можно забыть изменить тот или иной участок. Логика без строго выраженной формы размывается по кодовой базе и можно забыть о её наличие в том или ином участке.

Итак, дублирование плохо потому, что:

1. Сложней вносить изменения в программу ;

2. Безосновательно увеличивает кодовую базу ;
3. Дубли могут вносить новые аспекты в выполнение, которые будут скрыты от понимания из-за совпадения большей части кода (вспомнить детские ребусы – найди 10 отличий).

Что считается дубликатом? Код можно назвать дублирующимся по нескольким причинам. Так, например, в простейшем смысле, код дублируется, если происходит посимвольное совпадение. Но это не единственный случай. Код может дублироваться алгоритмически, при этом операции могут происходить над разными данными и с разными константами. Такое находить по началу сложней, и наша задача к этому привыкнуть.

Избавиться от дубликатов достаточно просто. Нам нужно извлечь метод (функцию), а все различающиеся данные вынести в параметры оставив внутри общий алгоритм.

Сделать программу понятней

Это одна их самых важных задач программиста. Не всегда, но зачастую. У нас командная работа, а также наша работа базируется на разработках других программистов. И если бы API их разработок не было

понятным, у нас возникало бы много трудностей при разработке.

Каждый раз, когда вы используете какую-нибудь библиотеку или даже базовые возможности языка C#, который в большей степени обеспечивается BCL (Base Class Library), задумайтесь о том, что это тоже некоторая кодовая база, с которой вы работаете и программисты, которые её разрабатывали, учитывали то, что вы будете работать с этим. И чем лучше они это делали, тем проще и комфортней работать вам.

При разработке своего кода учитывайте, что вы, скорей всего, забудете о том, как он работает и вам придётся разбираться с ним снова. И очень хорошо было бы упростить жизнь будущему себе. Я также надеюсь, что вы в ближайшее время начнете работать в команде и на основе вашего кода будут разрабатывать свои решения ваши коллеги. Вам повезет, если это будут мудрые, живущие в гармонии с собой и миром люди, транслирующие в мир любовь.

И не повезёт, если это будут психопаты, знающие, где вы живёте.

Есть один метод, который позволяет ответить на вопрос: “Стал ли мой код понятней?”, он заключается в задаче вопросов себе или другим людям, плохо

знакомым с кодом, но знакомым со спецификой вашей программы.

Можно задавать такие вопросы:

1. Сколько времени прошло от момента перевода взгляда на код, до момента когда была найдена нужная точка?
2. Читается ли код также хорошо, как книга писателя прозы средней руки?
3. Можно ли читать код не мотая головой вверх и вниз? Понятен ли он вне контекста или на сколько этот контекст важен?

Сокрыть структуру данных и тонкости работы с ней

Вы должны были заметить, что у нас есть некая сущность товара, которая описывает его стоимость, название и доступный остаток. Она представляется тремя массивами, которые не объединены друг с другом. А это значит, что при работе с товарами мы можем добавить название товара, но забыть добавить ему количество, что приведет к различным ошибкам. Более детально мы это проработаем дальше, сейчас же хотелось спрятать эти массивы и дать более очевидные методы работы с базой товаров.

Избавиться от дублирующегося кода

Мы будем препарировать листинг *WendingMachine/ArraysAndcycles/Program.cs*, который представлен на предыдущих страницах. Полностью я его приводить сейчас не буду из-за желания сэкономить бумагу и ваши деньги.

Наш основной инструмент в этот раз – метод (функция). Мы попробуем представить нашу программу как набор методов связанных друг с другом. Этим мы сразу же попробуем избавиться от дублирующегося кода. На методы мы будем также разбивать некоторые участки, а давая получившимся методам понятные имена, мы сделаем понятней весь наш код.

Чтобы победить дублирующийся код, нужно:

1. Найти альфа дублирующийся код;
2. Подойти к нему уверенно;
3. Укусить его за ушко;

После этого весь остальной дублирующийся код станет следовать за вами и вы сможете вывести его куда подальше.
Шутки шутками, но нам действительно нужно найти дублирующийся код. И не только альфа, а весь. И не кусать его, а выделять функцию. Если читать наш

листинг предыдущей главы, то бросается следующий блок кода.

Фрагмент 2.10

```
switch (payment)
{
    case PaymentType.Coins:
        for(int i = 0; i <
coinsValues.Length; i++)
        {
            Console.WriteLine($"Сколько монет
номиналом {coinsValues[i]} вы хотите
внести?");
            int count = 0;
            while
(!int.TryParse(Console.ReadLine(),
                                    out
count))
            {
                Console.WriteLine("Вы ввели
не число!");
            }
            coinsQuantity[i] += count;
            balance += count *
coinsValues[i];
        }
        break;
    case PaymentType.Card:
        Console.WriteLine("Сколько снять с
вашей карты?");
```

```
        int balanceDelta = 0;
        while
(!int.TryParse(Console.ReadLine(),
                out balanceDelta))
        {
            Console.WriteLine("Вы ввели не
число!");
        }
        balance += balanceDelta;
        Console.WriteLine("Баланс успешно
пополнен");
        break;
    default:
        break;
}
```

Обратите внимание на содержимое блоков case. У нас дублируется логика, которая состоит из цикла while. Она представляет из себя следующее:

1. Запросить у пользователя строку;
2. Удостоверится, что эта строка – число;
3. Если это так, то использовать эту строку как число далее;
4. Если это не так, запросить строку ещё раз и вывести поясняющее сообщение;

Дубликат 1.1

```
while (!int.TryParse(Console.ReadLine(),
```

```
                        out count))
{
    Console.WriteLine("Вы ввели не число!");
}
```

Дубликат 1.2

```
while (!int.TryParse(Console.ReadLine(),
        out balanceDelta))
{
    Console.WriteLine("Вы ввели не число!");
}
```

Как вы видите, разница этих дубликатов только в том, какая переменная будет использоваться для хранения результате. В данном случае нам не важно как в дальнейшем будет использоваться число.

Ответственность этого блока – чтение строки и конвертация в число, если это возможно, а если нет, то запрос новой строки. И в рамках этой обязанности мы будем работать.

В первую очередь мы поместим один из дубликатов в метод. Для начала мы его создадим, я его определяю в классе Program.

Фрагмент 2.11

```
private static int ReadInt()
```

```
{
    int result = 0;

    while (!int.TryParse(Console.ReadLine(),
                out result))
    {
        Console.WriteLine("Вы ввели не
число!");
    }

    return result;
}
```

У меня получился метод ReadInt, который служит для считывания числа. Давайте посмотрим как будет выглядеть код при использовании этого метода. Сейчас мы заменим два дубликата из фрагмента 10 и получим Фрагмент 2.12, который является частью полного листинга в конце этого раздела.

Фрагмент 2.12

```
switch (payment)
{
    case PaymentType.Coins:
        for (int i = 0; i <
coinsValues.Length; i++)
        {
            Console.WriteLine($"Сколько монет
номиналом {coinsValues[i]} вы хотите
```

```
внести?");
            int count = ReadInt();
            coinsQuantity[i] += count;
            balance += count *
coinsValues[i];
        }
        break;
    case PaymentType.Card:
        Console.WriteLine("Сколько снять с
вашей карты?");
        int balanceDelta = ReadInt();
        balance += balanceDelta;
        Console.WriteLine("Баланс успешно
пополнен");
        break;
    default:
        break;
}
```

Теперь, вместо дублирования логики мы просто вызываем один и тот же метод, который делает грязную работу за нас. Также мы уже затрагиваем тему понятности кода, разграничивая ответственности и не смешивая их мы получаем читаемый код.

Сейчас мы освободили логику команды от обязанностей чтения данных с консоли, но всё ещё сохранили зависимость логики от консоли. Это не очень хорошо и дальше мы это разберём, пока что

нас интересуют дубликаты.
Помните я изливал свою душу о том, что не хочу дублировать код? Тогда речь шла про вот этот кусочек:

Фрагмент 2.13

```
int id = Convert.ToInt32(rawData[0]);
int count = Convert.ToInt32(rawData[1]);
```

rawData – это массив строк. Тут задача превратить два элемента этого массива в значения двух переменных числового типа. Как мы можем это сделать? Сейчас мы пользуемся методом ToInt32 класса Convert.

Он нас устраивает лишь до того момента, пока в ячейках массива есть подходящие числа в виде строки. Тогда всё хорошо и метод произведет конвертацию. Но если там не корректные данные, метод выбросит исключение.

Что делать?

Использовать метод int.TryParse, который работает немного иначе. Он также производит конвертацию, но если она не удалась, то не выбрасывается исключение, а возвращается false.

```
public static bool TryParse(string s, out int
result)
```

Параметры

- s
    - Type: System.String
    - Строка, содержащая преобразуемое число.
- result
    - Type: System.Int32
    - При возвращении этим методом содержит 32-разрядное целочисленное значение со знаком, эквивалентное числу, содержащемуся в параметре s, если преобразование выполнено успешно, или нуль, если оно завершилось сбоем. Преобразование завершается сбоем, если параметр s равен null или String.Empty, не находится в правильном формате или представляет число меньше MinValue или больше MaxValue. Этот параметр передается неинициализированным; любое значение, первоначально предоставленное в объекте result, будет перезаписано.

Возвращаемое значение:

Type: System.Boolean

Значение true, если параметр s успешно преобразован; в противном случае — значение false.

Использование этого метода всегда превращается в несколько строк кода. Давайте попробуем воспользоваться им для преобразования первой ячейки массива.

Фрагмент 2.14

```
int id = 0;
if(!int.TryParse(rawData[0], out id))
{
    Console.WriteLine("Ошибка в параметре id,
он должен быть числом");
    break;
}
```

Если распространить этот код на вторую ячейку, то ситуация только ухудшится. Мы, понимая это, сделаем опережающий шаг и сразу же выделим метод. Он опять-таки будет в классе Program.

Фрагмент 2.15

```
private static bool MapParameter(string[]
rawParams,
                                 out int
containter,
```

```
BuyGoodParameter parameter)
{
    int index = (int)parameter;
    string name =
Enum.GetName(typeof(BuyGoodParameter),
parameter);

    if (!int.TryParse(rawParams[index], out
containter))
    {
        Console.WriteLine($"Ошибка в
параметре {name}, он должен быть числом");
        return false;
    }

    return true;
}
```

Получился интересный метод MapParameter. С его помощью мы не совсем избавляемся от дублирующегося кода, давайте посмотрим в место его применения и потом начнём разбор того, что произошло.

Фрагмент 2.16

```
int id = 0;
if(!MapParameter(rawData, out id,
BuyGoodParameter.Id))
{
    break;
```

```
}

int count = 0;
if (!MapParameter(rawData, out count,
BuyGoodParameter.Count))
{
    break;
}
```

Часть дублирующегося кода мы действительно скрыли в методе. Но всё-таки дубляж остался и пока что мы от него избавиться не можем. Один из вариантов – использование декларативного подхода к программированию с помощью рефлексии. Этим мы займёмся в самом конце этого проекта, после того как пройдёмся по другим темам.

И всё же код мы сделали по-лучше. Давайте вернемся к фрагменту 15, который содержит метод MapParameter. Он прост, с помощью него мы можем связать некий входной параметр нашей команды с переменной с типом int. В имени метода не используется уточнения, что мы ожидаем именно тип int, а вместо этого используется тип выходного параметра (параметра помеченного модификатором out). Этот параметр более красноречив нежели имя метода.

Итак, методу на вход мы даём:

- Аргументы, которые вводил пользователь, разбитые на элементы массива;
- Переменная, в которую нужно поместить значение параметра (в нашем случае с конвертацией в int);
- Параметр, с которым мы работаем, значение представлено в виде enum;

В ответ метод возвращает булевое значение true – если нет никаких проблем и false если что-то пошло не так. Я думаю, вы также заметили, что теперь при обращении к кому-нибудь параметру мы используем специально созданный тип BuyGoodParameter.

Фрагмент 2.17

```
enum BuyGoodParameter
{
    Id = 0,
    Count = 1
}
```

Сам шаг, я думаю, вам понятен, мы выделили большую часть логики в метод и сделали тип перечисления для более четкого представления параметров. Я думаю, многим читателям также интересна техническая реализация метода MapParameter.

Там есть два интересных момента.

Во-первых, мы можем использовать кастинг к типу int по отношению к типу перечисления. В результате будет какое-то определённое число, в нашем случае, если в метод придёт значение BuyGoodParameter.Id, то в результате кастинга будет значение 0. А если придёт значение BuyGoodParameter.Count, то будет значение 1.

Во-вторых, у нас в распоряжении есть метод Enum.GetName, который возвращает читаемое представление значения перечисления. Т.е при значении BuyGoodParameter.Id он вернёт строку “Id”. Крайне удобно.

Это всё? Нет, давайте заглянем дальше, посмотрите на это выражение.

Фрагмент 2.18

```
if (balance >= prices[id] * count)
{
    balance -= prices[id] * count;
    availableQuantity[id] -= count;
}
```

Здесь у нас дублируется умножение цены товара на приобретаемое количество. На первый взгляд это все безобидно, но в таком дубляже зачастую и кроются ошибки, потому что мозг по-началу и концу

строки быстро делает вывод, что они идентичны, хотя это может быть не так.

Сначала мы попробуем заменить выражение на переменную. Точнее оставим одно выражение и его результат поместим в переменную. Нам это подсказывает здравый смысл – зачем вычислять два раза, когда можно один?

Фрагмент 2.19

```
int totalPrice = prices[id] * count;

if (balance >= totalPrice)
{
    balance -= totalPrice;
    availableQuantity[id] -= count;
}
```

Всё ли мы сделали правильно? И да и нет. Есть один из классических рефакторингов – замена переменной вызовом метода.

Выглядит он вот так:

Фрагмент 2.20

```
if (balance >= GetTotalPrice(prices[id],
count))
{
```

```
    balance -= GetTotalPrice(prices[id],
count);
    availableQuantity[id] -= count;
}
```

И конечно же нам нужен метод GetTotalPrice, выглядит она вот так:

Фрагмент 2.21

```
private static int GetTotalPrice(int price,
int count)
{
    return price * count;
}
```

Да мы сделали метод, который рассчитывает стоимость заказа на основе таблицы цен и количества товара. Он имеет смысл, но немного не в том формате, в котором он сейчас. То что мы сейчас сделали, раскрывает нам одну важную мысль – у нас есть товар и у каждого товара есть набор свойств. Подумайте над тем, что эту мысль нужно выразить более строго в коде.

В той общей организации, которая у нас сейчас, сделанный нами метод очень громоздкий. И он на самом-то деле ничего не изменил. Фактически, мы сделали код немного понятней, навесив читаемую

метку на выражение. Но не избавились от дублирования.

Также суть выделения методов не только в том, чтоб вешать ярлыки на операции, но и в том, чтобы изменять уровень абстракции и сокрыть всякие ненужные детали.

Учитывая всё это, я бы остановился в итоге на следующем варианте:

Фрагмент 2.22

```
int totalPrice = GetTotalPrice(prices[id],
count);

if (balance >= totalPrice)
{
    balance -= totalPrice;
    availableQuantity[id] -= count;
}
```

На этом мы заканчиваем избавляться от дублирующегося кода. Дальше нас ждёт не менее важная задача – сделать программу понятней и сокрыть структуру данных.

Сокрыть структуру данных и сделать программу понятней

Сокрыв структуру данных я сделаю программу понятнее и безопаснее в обращении. Что я имею виду под структурой данных? Смотрите, у нас есть товар и хранилище товаров. Всё это организовано как ряд массивов, которые связанны друг с другом. Каждый массив хранит в себе данные определённого свойства товара. Массив обслуживает все товары.

Например, массив цен хранит цены всех товаров. А массив имён – имена. Ячейки под одним индексом в этих массивах относятся к одному товару. Так, например, вторая ячейка массива цен хранит цену второго товара. А вторая ячейка, массива имен, хранит имя второго товара.

	names	prices	availableQuantity
0	Шоколадка	70	5
1	Газировка	60	2

Работаем мы с такой структурой следующим образом. В этом примере я вывел всю информацию о первом товаре.

Фрагмент 2.23

```
Console.WriteLine($"Имя - {names[0]}");
Console.WriteLine($"Цена - {prices[0]}");
Console.WriteLine($"Остаток -
{availableQuantity[0]}");
```

Что нам нужно сделать с этим? Гораздо лучше здесь будут смотреться следующие функции: GetName, GetPrice, GetAvailableQuantity которым мы будем передавать Id товара в ответ будет получать нужное нам свойство.

Эти функции будут работать с глобальными переменными, которые будут представлены статическими полями класса Program.

Фрагмент 2.24

```
class Program
{
    private static int balance = 0;
    private static int[] coinsQuantity = { 0,
0, 0, 0 }; //1, 2, 5, 10
    private static int[] coinsValues = { 1,
2, 5, 10 };
```

```
    private static string[] names = {
"Шоколадка", "Газировка" };
    private static int[] prices = { 70, 60 };
    private static int[] availableQuantity =
{ 5, 2 };
    private static PaymentType payment =
PaymentType.Card;

    static void Main(string[] args)
    {
...
```

Я поднял все переменные из начала метода Main в поля класса Program. Это не совсем правильно делать. Дело в том, что есть концепция чистых функций, она очень полезная. Суть её в том, что поведение функции должно быть детерминировано и не иметь побочных эффектов. Также при одних и тех же входных значения, она должна возвращать одни и те же значения. Имеется ввиду, что она возвращает не то же самое, что ей прислали, а то что выходное значение строго привязано ко входному.

Здесь же мы собираемся сделать метод (функцию) которая будет работать с глобальными значениями. Это означает, что если я ей передам условный 0, то она может вернуть 1 а может 2. Это зависит от глобальных значений.

Я заметил что это не совсем правильно. Действительно не чистых функций стоит избегать. Но не в контексте ООП. В контексте ООП, если метод зависит от состояния объекта – это нормально, и чаще всего именно так и нужно сделать.

Мы сейчас ведём нашу программу в сторону ООП дизайна, поэтому будем использовать глобальные значения. Но предупреждаю, что в дальнейшем мы полностью откажемся от глобальных значений в пользу состояния объекта. Поэтому не стоит в будущем делать статические поля ссылаясь на этот материал.

И так, у нас есть структура данных и нам нужно сделать к ней фасад для комфортной работы. Представлен он будет следующим набором функций.

Фрагмент 2.25

```
private static string GetName(int id)
{
    if(id < 0 || id > names.Length) throw new
ArgumentOutOfRangeException("id");

    return names[id];
}

private static int GetPrice(int id)
{
```

```
    if(id < 0 || id > names.Length) throw new
ArgumentOutOfRangeException("id");

    return prices[id];
}

private static int GetAvailableQuantity(int
id)
{
    if(id < 0 || id > names.Length) throw new
ArgumentOutOfRangeException("id");

    return availableQuantity[id];
}

private static bool IsAvailableInQuantity(int
id, int count)
{
    return count < 0 || count >
GetAvailableQuantity(id);
}
```

Ничего не заметили? Правильно, дублирующийся код.

Попробуй исправить этот момент самостоятельно, не подглядывая на следующий исправленный фрагмент.

Фрагмент 2.26

```
private static bool Exist(int id)
{
    return id > 0 && id < names.Length;
}

private static void ValidateId(int id)
{
    if (!Exist(id))
    {
        throw new
ArgumentOutOfRangeException("id");
    }
}

private static string GetName(int id)
{
    ValidateId(id);
    return names[id];
}

private static int GetPrice(int id)
{
    ValidateId(id);
    return prices[id];
}

private static int GetAvailableQuantity(int
id)
{
    ValidateId(id);
    return availableQuantity[id];
```

```
}
```

У нас добавился метод Exist который позволяет проверить существует ли определённый товар. И вспомогательный метод ValidateId который выбрасывает исключение если товара нет. Он нужен в методах которые возвращают свойство товара. В этих методах также можно избавиться от постоянного вызова ValidateId, но вызовы уйдут сами как только мы перейдем к использованию классов и объектов.

Но в самом конце этой главы мы попадем в сходную ситуацию, и поборим сложность с помощью аспектно ориентированного программирования.

Весь сок проделанных действий будет виден когда мы начнём получившимися методами пользоваться. Давайте посмотрим на команду BuyGood в текущем состояние.

Фрагмент 2.27

```
else if (command.StartsWith("BuyGood"))
{
    //Разбиение строки на единицы данных
    string[] rawData =
command.Substring("BuyGood ".Length).Split('
');

    //Сопоставление этих данных с переменными
```

```
(и их типами)
    if (rawData.Length != 2)
    {
        Console.WriteLine("Неправильные
аргументы команды");
        break;
    }

    int id = 0;
    if(!MapParameter(rawData, out id,
BuyGoodParameter.Id))
    {
        break;
    }

    int count = 0;
    if (!MapParameter(rawData, out count,
BuyGoodParameter.Count))
    {
        break;
    }

    //Проверка корректности этих данных на
основе текущего состояния модели.
    if (id < 0 || id >= names.Length)
    {
        Console.WriteLine("Такого товара
нет");
        break;
    }
```

```
    if (count < 0 || count >
availableQuantity[id])
    {
        Console.WriteLine("Нет такого
количества");
        break;
    }

    //Выполнение
    int totalPrice =
GetTotalPrice(prices[id], count);

    if (balance >= totalPrice)
    {
        balance -= totalPrice;
        availableQuantity[id] -= count;
    }
}
```

Пора препарировать этот код. Воспользовавшись нашими методами мы получим следующее.

Фрагмент 2.28

```
else if (command.StartsWith("BuyGood"))
{
    //Разбиение строки на единицы данных
    string[] rawData =
command.Substring("BuyGood ".Length).Split('
');
```

```
    //Сопоставление этих данных с переменными
(и их типами)
    if (rawData.Length != 2)
    {
        Console.WriteLine("Неправильные
аргументы команды");
        break;
    }

    int id = 0;
    if(!MapParameter(rawData, out id,
BuyGoodParameter.Id))
    {
        break;
    }

    int count = 0;
    if (!MapParameter(rawData, out count,
BuyGoodParameter.Count))
    {
        break;
    }

    //Проверка корректности этих данных на
основе текущего состояния модели.
    if (Exist(id))
    {
        Console.WriteLine("Такого товара
нет");
        break;
```

```
    }

    if (IsAvailableInQuantity(id, count))
    {
        Console.WriteLine("Нет такого
количества");
        break;
    }

    //Выполнение
    int totalPrice =
GetTotalPrice(GetPrice(id), count);

    if (balance >= totalPrice)
    {
        balance -= totalPrice;
        availableQuantity[id] -= count;
    }
}
```

Вам может показаться, что поменялось мало что. Но на самом деле мы проделали важные изменения, которые станут более очевидными после перехода на объектно ориентированный дизайн.

Сейчас при работе с основной логикой программы нет никакого указания на то, что мы работаем с массивом. Вы вполне можете прийти в будущем к тому, что данные о товарах будут браться с базы

данных и в таком случае методы будут являться оберткой над ней.

Привычка работать с данными не напрямую сослужит вам хорошую службу при работе с ООП так, как там вам будут доступные полиморфные типы, которые позволяют вам на самом деле заменять источник данных и сокрытую структу без изменений остальной программы.

Давайте разобьем основной метод Main чтобы можно было быстро окинуть его взглядом и понять что происходит.

Я считаю, его можно разбить на следующие методы:

Фрагмент 2.29

```
private static string ReadCommand()
{
    Console.WriteLine("Введите команду:");
    return Console.ReadLine();
}

private static void ExecuteCommand(string
command)
{
    if (command == "AddMoney")
    {
        ...
```

```
    }
    else if (command == "GetChange")
    {
        ...
    }
    else if (command.StartsWith("BuyGood"))
    {
        ...
    }
    else
    {
        Console.WriteLine("Команда не
определена");
    }

}
```

В результате метод Main превратится в следующее:

Фрагмент 2.30

```
static void Main(string[] args)
{
    Console.WriteLine($"Имя - {names[0]}");
    Console.WriteLine($"Цена - {prices[0]}");
    Console.WriteLine($"Остаток -
{availableQuantity[0]}");

    while (true)
    {
        Console.Clear();
```

```
        Console.WriteLine($"Баланс
{balance}");

        string command = ReadCommand();
        ExecuteCommand(command);

        Console.ReadKey();
    }
}
```

Теперь программу легче читать, воспринимать и дополнять. Вы можете со мной не согласится, и к сожалению я не смогу вам возразить. То, что мы сделали не имеет большого утилитарного смысла, и нужна скорей для тренировки перед ООП. Также, нет никаких чётко измеримых показателей которые бы говорили, что получившийся в итоге код лучше того, что был.

Но основываясь на своём опыте, я бы предпочёл работать с итоговым вариантом.

Итог.

WendingMachine/FunctionalComposition/Program.cs

```
using System;

namespace FunctionalComposition
```

```
{
    class Program
    {
        private static int balance = 0;
        private static int[] coinsQuantity =
{ 0, 0, 0, 0 }; //1, 2, 5, 10
        private static int[] coinsValues = {
1, 2, 5, 10 };
        private static string[] names = {
"Шоколадка", "Газировка" };
        private static int[] prices = { 70,
60 };
        private static int[]
availableQuantity = { 5, 2 };
        private static PaymentType payment =
PaymentType.Card;

        static void Main(string[] args)
        {
            while (true)
            {
                Console.Clear();
                Console.WriteLine($"Баланс
{balance}");

                string command =
ReadCommand();
                ExecuteCommand(command);

                Console.ReadKey();
            }
```

```
        }

        private static string ReadCommand()
        {
            Console.WriteLine("Введите
команду:");
            return Console.ReadLine();
        }

        private static void
ExecuteCommand(string command)
        {
            if (command == "AddMoney")
            {
                switch (payment)
                {
                    case PaymentType.Coins:
                        for (int i = 0; i <
coinsValues.Length; i++)
                        {

Console.WriteLine($"Сколько монет номиналом
{coinsValues[i]} вы хотите внести?");
                            int count =
ReadInt();

                            coinsQuantity[i]
+= count;

                            balance += count
* coinsValues[i];
                        }
                        break;
```

```
                    case PaymentType.Card:

Console.WriteLine("Сколько снять с вашей
карты?");
                        int balanceDelta =
ReadInt();
                        balance +=
balanceDelta;

Console.WriteLine("Баланс успешно пополнен");
                        break;
                    default:
                        break;
                }
            }
            else if (command == "GetChange")
            {
                balance = 0;
            }
            else if
(command.StartsWith("BuyGood"))
            {
                //Разбиение строки на единицы
данных
                string[] rawData =
command.Substring("BuyGood ".Length).Split('
');

                //Сопоставление этих данных с
переменными (и их типами)
                if (rawData.Length != 2)
```

```
                {

Console.WriteLine("Неправильные аргументы
команды");
                    return;
                }

                int id = 0;
                if (!MapParameter(rawData,
out id, BuyGoodParameter.Id))
                {
                    return;
                }

                int count = 0;
                if (!MapParameter(rawData,
out count, BuyGoodParameter.Count))
                {
                    return;
                }

                //Проверка корректности этих
данных на основе текущего состояния модели.
                if (Exist(id))
                {
                    Console.WriteLine("Такого
товара нет");
                    return;
                }

                if (IsAvailableInQuantity(id,
```

```
count))
                {
                    Console.WriteLine("Нет
такого количества");
                    return;
                }

                //Выполнение
                int totalPrice =
GetTotalPrice(GetPrice(id), count);

                if (balance >= totalPrice)
                {
                    balance -= totalPrice;
                    availableQuantity[id] -=
count;
                }
            }
            else
            {
                Console.WriteLine("Команда не
определена");
            }

        }

        private static bool Exist(int id)
        {
            return id > 0 && id <
names.Length;
        }
```

```
        private static void ValidateId(int
id)
        {
            if (!Exist(id))
            {
                throw new
ArgumentOutOfRangeException("id");
            }
        }

        private static string GetName(int id)
        {
            ValidateId(id);
            return names[id];
        }

        private static int GetPrice(int id)
        {
            ValidateId(id);
            return prices[id];
        }

        private static int
GetAvailableQuantity(int id)
        {
            ValidateId(id);
            return availableQuantity[id];
        }

        private static bool
```

```
IsAvailableInQuantity(int id, int count)
        {
            return count < 0 || count >
GetAvailableQuantity(id);
        }

        private static int GetTotalPrice(int
price, int count)
        {
            return price * count;
        }

        private static int ReadInt()
        {
            int result = 0;

            while
(!int.TryParse(Console.ReadLine(),
                           out result))
            {
                Console.WriteLine("Вы ввели
не число!");
            }

            return result;
        }

        private static bool
MapParameter(string[] rawParams, out int
containter, BuyGoodParameter parameter)
        {
```

```
            int index = (int)parameter;
            string name =
Enum.GetName(typeof(BuyGoodParameter),
parameter);

            if
(!int.TryParse(rawParams[index], out
containter))
            {
                Console.WriteLine($"Ошибка в
параметре {name}, он должен быть числом");
                return false;
            }

            return true;
        }

    }

    enum BuyGoodParameter
    {
        Id = 0,
        Count = 1
    }

    enum PaymentType
    {
        Coins,
        Card
    }
}
```

Объектно ориентированный дизайн

Задача группы проектировщиков — создать иллюзию простоты.
Буч Г.

Какие сущности вы видите сейчас в коде? В предыдущей главе мы попытались разграничить код с помощью функций. Но это не дает нам достаточно точного описания объектов, с которыми мы работаем. Из-за этого мы не всегда можем понять четкой обязанности той части кода, с которой работаем.

В том маленьком примере, который мы сейчас используем, я бы не стал делать что-то ещё. Но нам нужно, во-первых, оправдать потребность в ООП, во-вторых, придумать такую задачу, которая задала бы вектор направления.

Начну с Анекдота:
“Заходит заказчик в бар и не знает, что выпить.”

Продолжу тем, что к вам подошёл заказчик (и вы не бармен) и просит сделать изменения в программе следующего порядка:

- Добавить новую команду, которая выводит список всех доступных команд;

- Сделать команду перехода в административный режим. После перехода в режим администратора все покупки становятся бесплатными.

Это могло бы стать вашей задачей. Но тогда вы могли бы накидать дополнительных методов и всё в вашей жизни было радужно. Но нет.

Ваша задача

Её изложил заказчик. Я лишь задам вам рамки:

- Вам нужно использовать классы, чтобы разделить программу на составляющие;
- Нельзя всю программу записать в два класса (WendingMachine и Good);
-

Возможное решение

У меня получился следующий набор классов:

- Program – наш корень компоновки, который строит весь граф объектов;
- Good – описывает товар;
- Order – описывает заказ;
- FreeOrder – описывает заказ, за который не нужно платить;
- Request – описывает запрос к машине;
- Router – преобразует запрос в команду, а также берёт на себя обязанность формирования заказа;
- WendingMachine – “помойка”, которая содержит в себе всё, что не захотелось на этом этапе выносить в другие классы. Содержит в себе товары и баланс;
- ConsoleCommandInput – читает строку с консоли и преобразует её в команду с помощью роутера;
- Commands:
    - AddMoney;
    - BuyGood;
    - GetChange;
    - ShowCommands;
    - Login;

Есть также у нас чистые абстракции, в виде интерфейсов:

- IOrder;

- ICommand;
- ICommandInput;

Они позволяют нам абстрагироваться во многих частях кода от того, откуда приходит запрос, какой тип заказа у нас и какую команду нужно выполнять.

Основная логика

Давайте заглянем в класс Program. Он позволяет нам понять то, как это всё соединяется. В нём мы работаем с самым верхним уровнем абстракции.

Листинг WendingMachine/OOP/Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        WendingMachine machine = new
WendingMachine(balance: 0,
                goods: new Good[]{
                    new Good("Шоколадка",
price: 70, count: 5),
                    new Good("Газировка",
price: 60, count: 2)
                }
        );
        ICommandInput input = new
ConsoleCommandInput(new Router(machine));

        while (true)
        {
            Console.Clear();
            Console.WriteLine($"Баланс
{machine.Balance}");

            var command = input.GetCommand();
```

```
            if (command == null)
            {
                Console.WriteLine("Команда не
распознана");
            }
            else
            {
                command.Execute();
            }
            Console.ReadKey();
        }
    }
}
```

Как вы видите, сначала мы конструируем наш вендинговый автомат, который содержит в себе товары и баланс. Далее мы собираем объект, который будет давать нам команды для выполнения.

То, что возвращает нам команды, прячется под абстракцией ICommandInput. В дальнейшем мы не будем знать, что работаем именно с консоли, теоретически мы можем сделать ввод команд с удаленного сервера или случайным, на основе текущих погодных условий.

Нужна ли нам эта абстракция в данный момент? Нет, но я исходил из своей любви к постоянному абстрагированию от средств ввода и вывода. Я хотел

сразу задать тон, что о том, что ввод идёт с консоли, знает только реализация ConsoleCommandInput. Ни в командах, ни в роутере этой информации быть не должно.

ConsoleCommandInput в конструктор мы передаем роутер, именно он будет заниматься созданием команд. Роутеру, в свою очередь, нужна вендинговая машина, так как команды будут конструироваться таким образом, что именно эта машина будет их точкой назначения и они будут над ней производить операции.

Далее ничего сложного. Бесконечный цикл, в котором мы получаем команду и запускаем её на выполнение. Абстракция команды – это всего лишь один метод – Execute.

Ранее, товары у нас размазывались по-нескольким массивам. Теперь же, за хранение товаров отвечает класс WendingMachine, а сам товар описывает класс Good.

Листинг WendingMachine/OOP/Good.cs

```
class Good
{
    public Good(string name, int price, int
count)
    {
```

```
        Name = name;
        Price = price;
        Count = count;
    }

    public string Name { get; private set; }
    public int Price { get; private set; }
    public int Count { get; set; }
}
```

Как вы видите, мы объединили данные о товаре в одном классе. И всё. Этот класс классический DTO (Data transfer object). Он не нужен ни для чего, кроме как для переноса информации. Вопрос только один, действительно ли количество товара должно располагаться в классе товара? На этот вопрос я хочу ответить в следующем разделе “Куда делся GoodStorage и OrderDeliver?”.

Теперь я хочу перейти к классу WendingMachine, он по сути содержит в себе все основные операции, которые в дальнейшем используются командами. Также, объект этого класса хранит в себе текущий баланс.

Листинг WendingMachine/OOP/WendingMachine.cs

```
class WendingMachine
{
```

```
    private Good[] _goods;

    public WendingMachine(int balance, params
Good[] goods)
    {
        _goods = goods;
        Balance = balance;
    }

    public int Balance { get; private set;  }

    public void AddBalance(int delta)
    {
        if (delta < 0) throw new
ArgumentOutOfRangeException("delta");

        Balance += delta;
    }

    public void DiscardBalance(int delta)
    {
        if (delta < 0 || Balance > delta)
throw new
ArgumentOutOfRangeException("delta");

        Balance -= delta;
    }

    public bool IsOrderPossible(IOrder order)
    {
        return order.IsAvailable &&
```

```
order.GetTotalPrice() <= Balance;
    }

    public bool TryProcessOrder(IOrder order)
    {
        if (IsOrderPossible(order))
        {
            Balance -= order.GetTotalPrice();
            order.Ship();
            return true;
        }
        else
        {
            return false;
        }
    }

    public bool IsContains(int id)
    {
        return id >= 0 && id < _goods.Length;
    }

    public Good GetFromId(int id)
    {
        if (!IsContains(id)) throw new
ArgumentOutOfRangeException("id");

        return _goods[id];
    }

}
```

Как вы уже могли заметить при чтение, у нас есть два метода IsOrderProcess и TryProcessOrder, которые нужны нам для выполнения заказа на продукты. Ответственность класса WendingMachine при обработке заказа заключается в том, что проверить его валидность с точки зрения доступного баланса и некоторой валидности с точки зрения заказа.

А потом просто снять деньги и доставить заказ. Причём за то, как доставлять заказ, ответственен сам заказ. Это может показаться немного странным. Но это, опять-таки, будет изложено в следующем разделе.

При работе с заказом мы работаем с абстракцией IOrder.

Листинг WendingMachine/OOP/IOrder.cs

```
interface IOrder
{
    bool IsAvailable { get; }

    int GetTotalPrice();
    void Ship();
}
```

Она нам нужна из-за того, что у нас есть 2 вида заказа: платный и бесплатный. Если мы хотим иметь

платный заказ, мы передаём в вендинговый аппарат объект класса Order.

Листинг WendingMachine/OOP/Order.cs

```
class Order : IOrder
{
    private Good _good;
    private int _count;

    public Order(Good good, int count)
    {
        if (count < 0) throw new
ArgumentOutOfRangeException();

        _good = good;
        _count = count;
    }

    public bool IsAvailable {
        get
        {
            return _count <= _good.Count;
        }
    }

    public int GetTotalPrice()
    {
        return _good.Price * _count;
    }
```

```
    public void Ship()
    {
        _good.Count -= _count;
    }
}
```

Он содержит в себе товар в одном экземпляре и его количество. И примитивную реализацию трёх методов. По сути абстракция не говорит нам о структуре заказа. Это может быть один товар в n-ом количестве.

А может быть целый ряд позиций. Мы можем спокойно добавлять новые типы заказов, не затрагивая вендинговый аппарат. Но эта абстракция будет не пригодная при ситуации, когда нам нужно визуализировать наш заказ. Например, вывести простой табличкой, что и в каком количестве заказано.

При такой реализации, нам нужно будет пересмотреть нашу абстракцию. Один из вариантов – воспользоваться паттерном Visitor.

Но вернёмся к обработке заказов. Итак, у нас есть обычный заказ, а есть бесплатный, который используется при входе в административный режим. Платный заказ рассмотрен выше, это класс Order. А вот бесплатный реализуется классом FreeOrder.

Листинг WendingMachine/OOP/FreeOrder.cs

```
class FreeOrder : IOrder
{
    private Good _good;
    private int _count;

    public FreeOrder(Good good, int count)
    {
        if (count < 0) throw new
ArgumentOutOfRangeException();

        _good = good;
        _count = count;
    }

    public bool IsAvailable
    {
        get
        {
            return _count <= _good.Count;
        }
    }

    public int GetTotalPrice()
    {
        return 0;
    }

    public void Ship()
    {
        _good.Count -= _count;
```

```
    }
}
```

Во-первых, отличие этого класса от предыдущего в том, что при реализации класс GetTotalPrice у нас возвращается всегда ноль, что приводит к тому, что при оплате такого заказа деньги не снимаются.

Во-вторых, учитывая лишь одно различие, у нас появляется множество дублирующегося кода. Как от него можно избавится? Решение этой задачи лежит в плоскости абстрактных классов.

Мы оставляем интерфейс IOrder, но вводим абстрактный класс SinglePositionOrder. В этом классе мы реализуем интерфейс IOrder лишь частично, делая метод GetTotalPrice абстрактным. Далее мы делаем две реализации: PayableOrder и FreeOrder. И как вы уже можете догадаться, делаем в них платную и бесплатную реализацию метода GetTotalPrice.

На вопрос “А почему бы просто не сделать виртуальный метод GetTotalPrice в классе Order и потом просто бы не наследоваться от Order классом FreeOrder и не переопределить в нем GetTotalPrice?”, я хочу ответить в разделе “Принцип подстановки Барбары Лисков на примере заказов”.

Роутинг и команды

Мы имеем основное ядро и дополнительную логику связи всего этого с пользовательским интерфейсом. У нас есть корневые абстракции ICommandInput и ICommand.

Листинг WendingMachine/OOP/ICommandInput.cs

```
interface ICommandInput
{
    ICommand GetCommand();
}
```

Листинг WendingMachine/OOP/ICommand.cs

```
interface ICommand
{
    void Execute();
}
```

Эти интерфейсы невероятно просты. По сути, первый позволяет нам абстрагироваться от того как и откуда приходят команды, а второй от реализации команды, которая пришла.

Теперь нам интересна реализация консольного ввода через класс ConsoleCommandInput.

Листинг WendingMachine/OOP/ConsoleCommandInput.cs

```
class ConsoleCommandInput : ICommandInput
{
    private Router _router;

    public ConsoleCommandInput(Router router)
    {
        _router = router;
    }

    public ICommand GetCommand()
    {
        string rawCommand =
Console.ReadLine();
        Request request =
ParseString(rawCommand);

        return
_router.CreateCommand(request);
    }

    private Request ParseString(string input)
    {
        string[] terms = input.Split(' ');
        int[] values = new int[0];

        if (terms.Length > 1)
        {
            values = new int[terms.Length -
1];
            for(int i = 1; i < terms.Length;
i++)
```

```
            {
                values[i-1] =
Convert.ToInt32(terms[i]);
            }
        }

        return new Request(terms[0], values);
    }
}
```

Здесь сразу в глаза бросается зависимость от некоторого роутера. Вся логика, по созданию команды по вводу, делегируется этой сущностью, которая является некой вариацией паттерна фабрика.

Ответственность именно ConsoleCommandInput состоит из того, чтобы прочитать строку с консоли и разобрать её в объект типа Request. Это происходит в методе ParseString. Сам по себе тип Request представляет следующее.

Листинг WendingMachine/OOP/Request.cs

```
class Request
{
    public Request(string command, int[]
values)
    {
        Command = command;
```

```
        Values = values;
    }

    public string Command { get; set; }
    public int[] Values { get; set; }

    public bool IsIncorectValuesCount(int
correct)
    {
        return correct != Values.Length;
    }
}
```

Запрос у нас состоит из команды и набора аргументов с типом int. А также методом, который помогает нам валидировать аргументы. В рамках типа Request мы можем только проверить соответствует ли текущее количество аргументов необходимому объему.

Для этого тут и нужен метод IsIncorectValueCount. Он делает немного обратное, он проверяет некорректное ли значение. Такой стиль выбран не просто так, в классе Router вы увидите причины, которые привели к этому.

Основная стратегия класса ConsoleCommandInput видна в методе GetCommand.

Фрагмент 2.31

```
public ICommand GetCommand()
{
    string rawCommand = Console.ReadLine();
    Request request =
ParseString(rawCommand);

    return _router.CreateCommand(request);
}
```

1. Считать строку;
2. Преобразовать её в запрос;
3. Создать из запроса команду;
4. Вернуть команду.

Третий шаг делегируется типу Router. Посмотрим на него поближе?

Листинг WendingMachine/OOP/Request.cs

```
class Router
{
    private WendingMachine _machine;
    private RouterState _state;

    public Router(WendingMachine machine)
    {
        _machine = machine;
        _state = new DefaultState(this);
    }

    public ICommand CreateCommand(Request
```

```
request)
    {
        switch (request.Command)
        {
            case "AddMoney":
                if
(request.IsIncorectValuesCount(1)) return
null;

                return new AddMoney(_machine,
request.Values[0]);
            case "GetChange":
                if
(request.IsIncorectValuesCount(0)) return
null;

                return new
GetChange(_machine);
            case "BuyGood":
                if
(request.IsIncorectValuesCount(2)) return
null;

                return new BuyGood(_machine,
_state.MakeOrder(request));
            case "ShowCommands":
                if
(request.IsIncorectValuesCount(0)) return
null;

                return new
```

```
ShowCommands("AddMoney", "GetChange",
"BuyGood", "ShowCommands");
            case "Login":
                if
(request.IsIncorectValuesCount(0)) return
null;

                return new Login(this);
            default:
                return null;
        }
    }

    public void Login()
    {
        _state = new AdminState(this);
    }

    public void Logout()
    {
        _state = new DefaultState(this);
    }

    abstract class RouterState
    {
        protected readonly Router Router;

        public RouterState(Router router)
        {
            Router = router;
        }
```

```
        public abstract IOrder
MakeOrder(Request request);
    }

    class DefaultState : RouterState
    {
        public DefaultState(Router router) :
base(router)
        {
        }

        public override IOrder
MakeOrder(Request request)
        {
            return new
Order(Router._machine.GetFromId(request.Value
s[0]), request.Values[1]);
        }
    }

    class AdminState : RouterState
    {
        public AdminState(Router router) :
base(router)
        {
        }

        public override IOrder
MakeOrder(Request request)
        {
```

```
            return new
FreeOrder(Router._machine.GetFromId(request.V
alues[0]), request.Values[1]);
        }
    }
}
```

Этот тип очень сложно описывать. Если что-то в вашей программе сложно описать, то скорей всего это нужно переписать. Я не берусь говорить, что в случае с этим кодом, это не так. Точнее, я это скажу, так как считаю, что сложность именно в идиоматической сложности.

Начнём с основного метода, который создает из запроса команду. Называется он CreateCommand и состоит из огромного свича, который строковое представление преобразует в объект заранее заданного типа. Правильно ли это? Не совсем. У нас есть каверзная команда – вывод всех команд. И, к сожалению, она никак не связана с этим свичом, из-за чего она может лгать.

Этот момент как бы раскрывает нам глаза на ту проблему, которую мы вызвали этим свичём. Мы уже провели большую работу и инкапсулировали команду в объект и спрятали всё под интерфейсом ICommand. Но мы всё ещё не имеем расширяемого механизма преобразования строки в объект команды.

Вкратце проблема этого свича в следующем:

- Список доступных команд захардкожен и не может расширяться без переписи кода;
- Дублируется код валидации количества аргументов;

Мы героически это исправим в разделе “Рефлексия и атрибуты на примере команд”.

Переведите свой взгляд на кейс команды BuyGood. Он интересен тем, что в конструктор, сходного по названию класса, нужно передать заказ. Посмотрите откуда заказ берётся. Заметили? Там мы обращаемся к некоторому текущему состоянию нашего роутера. И это состояние выдаёт нам заказ.

Состояние скрыто под абстракцией RouterState (самый низ предыдущего листинга). Это абстрактный класс с одним методом MakeOrder. Он определяется в производных классах DefaultState и AdminState. Первый возвращает платный заказ, а второй бесплатный.

Переход в это состояние происходит в методах Login и Logout. Вот так мы решили задачу того, что у нас есть два состояния: административное и обычное. И эти состояния находятся в роутере. Инициатором переключения состояний и является команда Login.

А теперь пора рассмотреть реализации всех команд. Всего их 5.

Листинг
WendingMachine/OOP/Commands/AddMoney.cs

```
class AddMoney : ICommand
{
    private WendingMachine _machine;
    private int _money;

    public AddMoney(WendingMachine machine,
int money)
    {
        _machine = machine;
        _money = money;
    }

    public void Execute()
    {
        _machine.AddBalance(_money);
    }
}
```

Листинг
WendingMachine/OOP/Commands/BuyGood.cs

```
class BuyGood : ICommand
{
    private WendingMachine _machine;
    private IOrder _order;
```

```
    public BuyGood(WendingMachine machine,
IOrder order)
    {
        _machine = machine;
        _order = order;
    }

    public void Execute()
    {
        _machine.TryProcessOrder(_order);
    }
}
```

Листинг
WendingMachine/OOP/Commands/GetChange.cs

```
class GetChange : ICommand
{
    private WendingMachine _machine;

    public GetChange(WendingMachine machine)
    {
        _machine = machine;
    }

    public void Execute()
    {

_machine.DiscardBalance(_machine.Balance);
    }
```

```
}
```

Листинг WendingMachine/OOP/Commands/Login.cs

```
class Login : ICommand
{
    private Router _router;

    public Login(Router router)
    {
        _router = router;
    }

    public void Execute()
    {
        _router.Login();
    }
}
```

Листинг WendingMachine/OOP/Commands/ShowCommands.cs

```
class ShowCommands : ICommand
{
    private string[] _commands;

    public ShowCommands(params string[]
commands)
    {
        _commands = commands;
```

```
    }

    public void Execute()
    {
        foreach (string command in _commands)
        {
            Console.WriteLine(command);
        }
    }
}
```

Всё, на этом исходный код закончился. Чего мы добились этим решением:

- Мы разделили логику программы на составные части, которые можем дорабатывать раздельно. При работе с мелкими частями, мы лучше понимаем контекст и нам сложней что-то сломать.
- Мы добавили швы, которые позволяют нам расширять функционал.
- Мы инкапсулировали команду в объект.

Куда делся GoodStorage и OrderDeliver?

В нашем примере вы могли заметить отсутствие какой-либо сущности, которая занималась бы доставкой заказа. Действительно, когда мы думаем о вендинговом аппарате, мы подразумеваем то, что физически весь товар размещен в каком-то хранилище и есть некая механическая система доставки.

Но в нашей реализации этого нет. Поэтому, мы смогли пойти на некоторые условности. Во-первых, ответственность об доставке товара мы отдали заказу. Во-вторых, доставка товара сводится к убавлению количества у определённого объекта.

Если бы разрабатывали чуть более расширенную систему, то у нас появилось бы два важных класса:

- GoodStorage;
- OrderDeliver.

Первый бы содержал в себе товары и их остаток, возможно с разбивкой по слотам, в которых они находятся. Второй занимался бы доставкой:

1. Снимал бы деньги с баланса аппарата;
2. Забирал товар с хранилища;
3. Контролировал бы передачу клиенту.

Принцип подстановки Барбары Лисков на примере заказов

Я напомню вам, что у нас есть дублирующийся код. Он находится в листинге класса “FreeOrder” и “Order”. По сути оба класса являются копиями друг друга с небольшими изменениями.

Это интересная ситуация, для которой я предложил следующее решение:

> “Решение этой задачи лежит в плоскости абстрактных классов. Мы оставляем интерфейс IOrder, но вводим абстрактный класс SinglePositionOrder. В этом классе мы реализуем интерфейс IOrder лишь частично, делая метод GetTotalPrice абстрактным. Далее мы делаем две реализации: PayableOrder и FreeOrder. И как вы уже можете догадаться, делаем в них платную и бесплатную реализацию метода GetTotalPrice.”
>
> Я чуть ранее

Прочитав это вы могли задаться вопросом: “А почему бы просто не сделать виртуальный метод GetTotalPrice в классе Order, а потом просто бы не унаследоваться от Order классом FreeOrder и не переопределить в нём GetTotalPrice?”.

Я пообещал дать ответ в этом разделе. И настало время ответить за свои слова. Основание моего решения находятся в одном из принципов “S.O.L.I.D.” А именно в принципе LSP – это принцип подстановки Барбары Лисков.

Его можно определить несколькими способами. Но в этот раз я хочу воспользоваться более эмпирическим описанием: “Если есть некий тип A и производный тип B, то мы должны иметь возможность использовать B вместо типа A, при этом поведение программы не должно меняться”. Грубо говоря, тест, который проходит тип A, должен проходить и тип B.

Этот принцип, как и все остальные, не являются обязательными для соблюдения. Даже больше, слепое и фанатичное следование им может привести вас к более худшим результатам, чем вообще отказ от них.

В данной ситуации мы решили последовать ему. И для этого нам нужно будет сделать два решения, и пустить их через призму этого принципа. Взглянем на решение на основе виртуальных методов и переопределения поведения типа Order.

Решение 1

```
class Order : IOrder
{
    private Good _good;
    private int _count;

    public Order(Good good, int count)
```

```
    {
        if (count < 0) throw new
ArgumentOutOfRangeException();

        _good = good;
        _count = count;
    }

    public bool IsAvailable {
        get
        {
            return _count <= _good.Count;
        }
    }

    public virtual int GetTotalPrice()
    {
        return _good.Price * _count;
    }

    public void Ship()
    {
        _good.Count -= _count;
    }
}

class FreeOrder : Order
{
    public FreeOrder(Good good, int count) :
base(good, count)
    {
```

```
    }

    public  override int GetTotalPrice()
    {
        return 0;
    }
}
```

Выглядит лаконично! Но это простота может быть обманчивой. Как вы заметили, я не хочу тут ничего чётко утверждать. Такое решение действительно простое и в некоторых ситуациях подходящее.

Но мы должны с вами понимать, что FreeOrder – это костыль, который мы подсовываем системе и она начинает работать так, как мы хотим. Это не такой явный костыль, как если бы, вместо команды добавления денег, использовали команду покупки с типом, который имеет отрицательную цену. И всё же мы ситуативно и несогласованно изменяем поведение системы.

Более корректное решение будет проведено через абстрактный класс, который содержит такую спецификацию, что заказ имеет один товар определенного количества. Но в нём нет определения платный ли это товар или бесплатный.

Решение 2

```
abstract class Order : IOrder
{
    protected readonly Good Good;
    protected readonly int Count;

    public Order(Good good, int count)
    {
        if (count < 0) throw new
ArgumentOutOfRangeException();

        Good = good;
        Count = count;
    }

    public bool IsAvailable {
        get
        {
            return Count<= Good.Count;
        }
    }

    public abstract int GetTotalPrice();

    public void Ship()
    {
        Good.Count -= Count;
    }
}

class PayableOrder : Order
{
```

```
    public PayableOrder(Good good, int count)
: base(good, count)
    {
    }

    public override int GetTotalPrice()
    {
        return Good.Price * Count;
    }
}

class FreeOrder : Order
{
    public FreeOrder(Good good, int count) :
base(good, count)
    {
    }

    public override int GetTotalPrice()
    {
        return 0;
    }
}
```

Разница в том, что мы не можем ставить бесплатный заказ там, где ожидается платный, так же как и обратное. Так что система будет работать корректно и нам сложней её сломать.

Вопрос с тестами тоже закрывается просто – мы не можем тестировать абстрактный класс без создания

конкретной реализации. Мы не имеем конкретного поведения в точке расчета цены, поэтому мы не можем как-то контролировать результат.

Технически мы можем попробовать тестами описать то, что число должно быть всегда больше или равно нулю. Но в таком тесте уже не описать, что при создание заказа с двумя шоколадками, его цена должна быть 60.

Именно то, что метод абстрактный, говорит о том, что стратегия формирования цены спрятана от нас. Если в некой подсистеме нам достаточно только того, что число положительное, то мы работаем именно с типом Order. Если же мы хотим иметь товар, цена которого имеет прямую зависимость только от того, какой товар и в каком количестве мы добавили, мы используем тип PayableOrder.

Таким разделением типов мы позволяем нам точней декларировать, например, функциональные члены. Точно задавая то, что мы хотим без догадок и предположений, которые ни на чём не основываются.

Стоит ли изменять тип параметра в методе TryProcessOrder

Учитывая вышесказанное, я хочу с вами присмотреться к методу TryProcessOrder из класса WendingMachine.

Фрагмент 2.32

```
public bool TryProcessOrder(IOrder order)
{
    if (IsOrderPossible(order))
    {
        Balance -= order.GetTotalPrice();
        order.Ship();
        return true;
    }
    else
    {
        return false;
    }
}
```

Напоминаю, что этот метод снимает баланс вендингового аппарата и делегирует доставку заказа обратно заказу. Внимание на себя обращает строка со снятием баланса. В ней мы всё-таки предполагаем, что заказ чего-то стоит.

Так ли это? Это сложный вопрос. Сначала спросим у абстракции действительно ли она говорит о том, что заказ может чего-то стоить?

Фрагмент 2.33

```
interface IOrder
{
    bool IsAvailable { get; }

    int GetTotalPrice();
    void Ship();
}
```

Метод GetTotalPrice описан таким образом, что он возвращает тип int. А это означает, что он может вернуть и ноль, и отрицательное число. Так верно ли предположение о том, что заказ чего-то стоит? Нет.

Поэтому, в классе WendingMachine мы не можем на это рассчитывать. И если бы мы хотели иметь дело только с заказами, которые предоставляют какую-либо ценность, мы бы изменили тип параметра с IOrder на PayableOrder, что давало бы нам гарантию стоимости товара.

Было бы очень неловко разрушать это всё одним виртуальным методом, который бы сделал платный заказ опять бесплатным.

Рефлексия на примере команд

Мы можем с вами вспомнить класс Router, который берёт на себя ответственность преобразования некоторого запроса в объект команды.

С помощью свича мы производим ассоциацию строки (название команды вводимой пользователем) с объектом, который берёт на себя обязанность непосредственного выполнения. Помимо этого, мы также предоставляем команде всё необходимое для выполнения.

Так, команда, с помощью конструктора, декларирует, что ей нужно для работы. Часть из того, что ей нужно, передается пользователем через консоль. А часть является частью системы и передается нашим роутером, а пользователь об этом не знает.

В том оформлении, которое есть у нас сейчас, есть ряд недостатков.

Фрагмент 2.33

```
public ICommand CreateCommand(Request
request)
{
    switch (request.Command)
    {
        case "AddMoney":
```

```
            if
(request.IsIncorectValuesCount(1)) return
null;

            return new AddMoney(_machine,
request.Values[0]);
        case "GetChange":
            if
(request.IsIncorectValuesCount(0)) return
null;

            return new GetChange(_machine);
        case "BuyGood":
            if
(request.IsIncorectValuesCount(2)) return
null;

            return new BuyGood(_machine,
_state.MakeOrder(request));
        case "ShowCommands":
            if
(request.IsIncorectValuesCount(0)) return
null;

            return new
ShowCommands("AddMoney", "GetChange",
"BuyGood", "ShowCommands");
        case "Login":
            if
(request.IsIncorectValuesCount(0)) return
null;
```

```
            return new Login(this);
        default:
            return null;
    }
}
```

Что в этом плохого? На самом деле ничего. В большинстве ситуаций этого будет вполне достаточно. Но в других случаях такое решение может быть не оптимальным. Во-первых, тут дублируется код, во-вторых, добавление новой команды приводит к изменению класса Router.

Мы можем избавиться от этого с помощью рефлексии. Рефлексия – это специальный механизм, который позволяет приложению “смотреть” в себя и работать с типами и их содержим, как с обычными объектами. Что даёт нам возможность, например, найти все классы, которые реализуют интерфейс ICommand, взять из них класс с определенным именем и создать из него объект, а далее вернуть ссылку на него.

Именно это мы и сделаем с вами. Мы не будем затрагивать ничего, кроме класса Router и все изменения будут только в нём (кроме команды ShowCommands).

Базой для нас является возможность динамически найти все типы в приложение, которые удовлетворяют нас.

Фрагмент 2.34

```
private readonly Type _commandBaseType =
typeof(ICommand);

...

private IEnumerable<Type> GetCommandsTypes()
{
    return AppDomain
        .CurrentDomain
        .GetAssemblies()
        .SelectMany(assembly =>
assembly.GetTypes())
        .Where(type =>
_commandBaseType.IsAssignableFrom(type))
        .Where(type => IsRealClass(type));
}

private Type GetCommandTypeByName(string
name)
{
    return GetCommandsTypes()
        .Where(type => type.Name == name)
        .FirstOrDefault();
}
```

```
private bool IsRealClass(Type testType)
{
    return testType.IsAbstract == false
           &&
testType.IsGenericTypeDefinition == false
           && testType.IsInterface == false;
}
```

С помощью метода GetCommandsType, мы получаем все типы, которые реализуют интерфейс ICommand, которые являются не абстрактным и не обобщенным классом.

А с помощью метода GetCommandTypeByName мы уже среди них выбираем только те, имя которых нам подходит. В нашем новом роутере теперь всё работает таким образом, что имя команды, которое вводит пользователь, является прямым указанием на то, какой класс эту команду реализует. Т.е для команды ShowCommands он будет искать класс с этим названием. И очень важно, чтобы этот класс реализовывал интерфейс ICommand.

Когда у нас есть информация о типе, мы можем сделать много интересных действий. Для нас важно одно – создать объект этого типа. Но чтобы его совершить, нужно решить следующие задачи:

1. Найти подходящий нам конструктор;
2. Подготовить аргументы для его вызова;
3. Вызвать его;

Фрагмент 2.35

```
private ICommand CreateInstance(Type type,
Request request)
{
    ConstructorInfo[] constructors =
type.GetConstructors(BindingFlags.Instance |
BindingFlags.Public);

    foreach (var ctor in constructors)
    {
        var args =
ResolveDependeciesAndMerge(ctor, request);
        if (args != null)
        {
            return
(ICommand)ctor.Invoke(args);
        }
    }

    return null;
}
```

За это у нас ответственен метод CreateInstance, который создает объект команды из заданного типа на основе пришедшего запроса (напоминаю, что в запросе находится данные для команды).

Сначала мы ищем все конструкторы объекта, которые являются публичными. Далее мы

перебираем их до тех пор, пока не сможешь найти тот, которому мы сможем дать всё необходимое. Как только мы его находим, мы его вызываем передавая данные для его параметров. В ответ он нам возвращает объект, который возвращаем и мы.

Благодаря этим методам нам удалось сократить метод CreateCommand и сделать роутер динамически подстраиваемым под уже имеющиеся команды. Если мы хотим добавить команду, нам нужно только создать класс и определить в нём публичный конструктор. Всё остальное за нас сделает построенная нами система.

Фрагмент 2.36

```
public ICommand CreateCommand(Request
request)
{
    var commandType =
GetCommandTypeByName(request.Command);
    if (commandType != null)
    {
        var instance =
CreateInstance(commandType, request);
        return instance;
    }
    else
    {
        return null;
    }
}
```

Разрешение зависимостей

При вызове конструктора нам нужно передать ему аргументы. Параметры конструктора формируют зависимости, которые нужное разрешить. Так, например, команда ShowCommands для своего выполнение требует ссылку на роутер, который предоставит ей список всех доступных команд для последующего вывода.

Листинг WendingMachine/CommandsSystem/ShowCommands.cs

```
class ShowCommands : ICommand
{
    private Router _router;

    public ShowCommands(Router router)
    {
        _router = router;
    }

    public void Execute()
    {
        foreach(var command in
_router.GetCommands())
        {
            Console.WriteLine(command);
        }
    }
```

```
}
```

Сам метод GetCommands находится в классе Router и не представляет ничего интересного. Он просто преобразует типы всех доступных команд в строку (по их имени) и выдает под интерфейсов IEnumerable<string>

Фрагмент 2.37

```
public IEnumerable<string> GetCommands()
{
    return GetCommandsTypes().Select(type =>
type.Name);
}
```

Интересует нас другое. Наш роутер может найти любой тип, который отвечает определённым требования. И у нас нет четких требований от чего может зависеть команда. Ей, для выполнения, может потребоваться самый широкий набор сущностей. Единственное, что мы знаем, это то, что все зависимости будут отражены через параметры конструктора. На самом деле это не обязательно, но то, что мы будем делать, основывается на этом факте.

Итак, у нас есть конструктор, который декларирует то, что нужно команде через параметры. Параметры

могут быть любого типа и нам нужно предоставить аргументы под эти параметры. Нам нужно выполнить следующие действия:

1. Если параметр типа int, то мы делаем аргументом значение из запроса (Request) по порядку;
2. Если параметр другого типа, отличного от примитивного (не string, byte, etc.), то мы разрешаем зависимость на основе таблицы, которая есть в роутере;

За это всё ответственен метод ResolveDependeciesAndMerge, который используется в методе CreateInstance. На вход этот метод получает конструктор и запрос, а далее пытается предоставить аргументы к каждому параметру. Если это удаётся – он их возвращает, если нет – он возвращает null.

Фрагмент 2.38

```
private object[]
ResolveDependenciesAndMerge(ConstructorInfo
constructor, Request request)
{
    List<object> args = new List<object>();
    Queue<int> requestArgs = new
Queue<int>(request.Values);

    foreach (var parameter in
```

```
constructor.GetParameters())
    {
        if
(_dependecies.TryGetValue(parameter.Parameter
Type, out object value))
        {
            args.Add(value);
        }
        else
        {
            if (requestArgs.Count == 0)
return null;

            args.Add(requestArgs.Dequeue());
        }
    }

    if (args.Count ==
constructor.GetParameters().Length)
    {
        return args.ToArray();
    }
    else
    {
        return null;
    }
}
```

Этому методу для работы также нужна таблица, которая тоже находится в классе Request.

Фрагмент 2.39

```
private Dictionary<Type, object>
_dependecies;

public Router(WendingMachine machine)
{
    ...

    _dependecies = new Dictionary<Type,
object>()
                    {
                        {
typeof(WendingMachine), _machine},
                        { typeof(Router),
this }
                    };
}
```

В ResolveDependeciesAndMerge мы имеем последовательность параметров и очередь аргументов из запроса. А также список аргументов, которые можно отправить в конструктор. Точнее, этот список нам нужно получить в результате.

Таблица нам помогает разрешать зависимость от определённого типа. Так, если параметр определён с типом WengingMachine и мы дадим ему ссылку на конкретный объект этого типа. Расширяя таблицу, мы

расширяем спектр зависимостей, которые можем передавать нашим командам.

Нам также стоит учитывать, что в конструкторе параметры могут определяется в разной последовательность. Например, оба эти конструктора мы должны обрабатывать корректно, несмотря на то, что параметры в разном порядке.

Фрагмент 2.40

```
public AddMoney(WendingMachine machine, int
money)
{
    _machine = machine;
    _money = money;
}

...

public AddMoney(int money, WendingMachine
machine)
{
    _machine = machine;
    _money = money;
}
```

Замечу, что мы говорим не про одновременное существование этих двух конструкторов (такого быть

не может), а про независимость от последовательности параметров.

Нужного нам поведения мы достигаем с помощью очереди. Для каждого параметра мы либо берём значение с таблицы, либо достаем из очереди, созданной на основе запроса.

Вот полный листинг нового роутера.

Листинг WendingMachine/CommandsSystem/Router.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;

namespace CommandsSystem
{
    class Router
    {
        private readonly Type
_commandBaseType = typeof(ICommand);

        private WendingMachine _machine;
        private RouterState _state;
        private Dictionary<Type, object>
_dependecies;

        public Router(WendingMachine machine)
```

```
        {
            _machine = machine;
            _state = new DefaultState(this);

            _dependecies = new
Dictionary<Type, object>()
                                {
                                    {
typeof(WendingMachine), _machine},
                                    {
typeof(Router), this }
                                };
        }

        public ICommand CreateCommand(Request
request)
        {
            var commandType =
GetCommandTypeByName(request.Command);
            if (commandType != null)
            {
                var instance =
CreateInstance(commandType, request);
                return instance;
            }
            else
            {
                return null;
            }
        }
```

```
        public void Login()
        {
            _state = new AdminState(this);
        }

        public void Logout()
        {
            _state = new DefaultState(this);
        }

        public IEnumerable<string>
GetCommands()
        {
            return
GetCommandsTypes().Select(type => type.Name);
        }

        private object[]
ResolveDependeciesAndMerge(ConstructorInfo
constructor, Request request)
        {
            List<object> args = new
List<object>();
            Queue<int> requestArgs = new
Queue<int>(request.Values);

            foreach (var parameter in
constructor.GetParameters())
            {
                if
(_dependecies.TryGetValue(parameter.Parameter
```

```
Type, out object value))
                {
                    args.Add(value);
                }
                else
                {
                    if (requestArgs.Count ==
0) return null;


args.Add(requestArgs.Dequeue());
                }
            }

            if (args.Count ==
constructor.GetParameters().Length)
            {
                return args.ToArray();
            }
            else
            {
                return null;
            }
        }

        private IEnumerable<Type>
GetCommandsTypes()
        {
            return AppDomain
                .CurrentDomain
                .GetAssemblies()
```

```
                .SelectMany(assembly =>
assembly.GetTypes())
                .Where(type =>
_commandBaseType.IsAssignableFrom(type))
                .Where(type =>
IsRealClass(type));
        }

        private Type
GetCommandTypeByName(string name)
        {
            return GetCommandsTypes()
                .Where(type => type.Name ==
name)
                .FirstOrDefault();
        }

        private bool IsRealClass(Type
testType)
        {
            return testType.IsAbstract ==
false
                    &&
testType.IsGenericTypeDefinition == false
                    && testType.IsInterface
== false;
        }

        private ICommand CreateInstance(Type
type, Request request)
        {
```

```
            ConstructorInfo[] constructors =
type.GetConstructors(BindingFlags.Instance |
BindingFlags.Public);

            foreach (var ctor in
constructors)
            {
                var args =
ResolveDependeciesAndMerge(ctor, request);
                if (args != null)
                {
                    return
(ICommand)ctor.Invoke(args);
                }
            }

            return null;
        }

        abstract class RouterState
        {
            protected readonly Router Router;

            public RouterState(Router router)
            {
                Router = router;
            }

            public abstract IOrder
MakeOrder(Request request);
        }
```

```
        class DefaultState : RouterState
        {
            public DefaultState(Router
router) : base(router)
            {
            }

            public override IOrder
MakeOrder(Request request)
            {
                return new
PayableOrder(Router._machine.GetFromId(reques
t.Values[0]), request.Values[1]);
            }
        }

        class AdminState : RouterState
        {
            public AdminState(Router router)
: base(router)
            {
            }

            public override IOrder
MakeOrder(Request request)
            {
                return new
FreeOrder(Router._machine.GetFromId(request.V
alues[0]), request.Values[1]);
            }
```

```
            }
        }
}
```

Позиционная ассоциация аргументов с параметрами

Сейчас, при определении того, какое значение из запроса в какой параметр будет отправлено, мы основываемся на их позиция. Так, если у нас есть условный конструктор с двумя параметрами с типом int и есть две значения в запросе, то первое уйдёт в первые параметр а второе – во второй.

Пока это нас вполне устраивает. Но мы не имеем возможности при вводе значения в консоли (или через любой другой ввод) указать то, что первое число должно уходить во второй параметр, а второй – в первый.

Это можно было бы достичь с помощью меток. Например, для выполнения команды BuyGood, можно было бы записывать такую строку:

BuyGood goodId:1 count:2

Вместо :

BuyGood 1 2

Чтобы сохранить поддержку ввода старого образца, мы могли бы сделать ещё одну сущность, которая преобразовывала бы обычный позиционный ввод в

именованные аргументы, которые мы могли бы в дальнейшем более точно ассоциировать с параметрами.

В классе Request появился бы примерно такой метод.

Фрагмент 2.41

```
public bool TryGetValue(string name, out int
value)
{

}
```

И мы бы им пользовались в классе Router в методе определения аргументов.

Фрагмент 2.42

```
private object[]
ResolveDependeciesAndMerge(ConstructorInfo
constructor, Request request)
{
    List<object> args = new List<object>();

    foreach (var parameter in
constructor.GetParameters())
    {
        if
(_dependecies.TryGetValue(parameter.Parameter
Type, out object value))
```

```
        {
            args.Add(value);
        }
        else
        {
            if
(request.TryGetValue(parameter.Name, out int
arg))
            {
                args.Add(arg);
            }
            else
            {
                return null;
            }
        }
    }

    if (args.Count ==
constructor.GetParameters().Length)
    {
        return args.ToArray();
    }
    else
    {
        return null;
    }
}
```

Наделяем команды описанием через атрибуты

На данный момент мы используем имя типа, который представляет команду как некий ключ, по которому мы можем её найти. Если пользователь вводит команду AddMoney, то мы буквально ищем одноименный класс. Это не всегда удобно и нам хотелось бы наделить команду некоторой описательной информацией.

Чего хочется добится:

1. Возможность давать команде имя;
2. Проверять на этапе компиляции что нет двух команд с одинаковым именем;

Один из возможных путей решение состоит в том, чтобы расширить интерфейс ICommand и добавить в него метод или свойство GetName. Этот вариант интересен и даже весь жизнеспособен. Но не в нашей ситуации. Во-первых, команду описывает тип, а не объект. Так что мы не можем говорить, что имя является частью объекта. Во-вторых, это затруднит статический анализ. В-третьих, не всегда подобные вещи являются частью абстракции ICommand.

Сейчас я хочу описать решение через атрибуты. Атрибуты – это специальных механизм C#, который позволяет нам добавлять декларативное описание к

различным сущностям. И впоследствии пользоваться этим описанием через механизм рефлексии.

Мы можем создавать как свои атрибуты, так и использовать уже готовые. Яркий пример атрибут Serializable, который позволяет пометить необходимость сериализации того или иного члена. Я хочу создать атрибут Command, который позволит мне добавить описание класса. В нашем случае – название команды.

Начнём с создания атрибута.

Листинг
WendingMachine/CommandsSystem/Attributes.cs

```
[AttributeUsage(AttributeTargets.Class,
AllowMultiple = true, Inherited = false)]
class CommandAttribute : Attribute
{
    public string CommandName;

    public CommandAttribute(string
commandName)
    {
        CommandName = commandName;
    }
}
```

Нам нужно определить класс CommandAttribute, который наследуется от Attribute. А также, нужно пометить получившийся класс атрибутом AttributeUsage, в котором мы указываем: к чему мы можем применять получившийся атрибут, можно ли применять его несколько раз и наследуется ли он. В нашем случае я указал, что один класс может быть связан с несколькими команда и то, что производные классы не являются связанными с этими же командами.

Теперь рассмотрим применение этого. Во-первых, вот так мы можем пометить класс.

Листинг WendingMachine/CommandsSystem/Commands/AddMoney.cs

```
[Command("AddMoney")]
class AddMoney : ICommand
{
    private WendingMachine _machine;
    private int _money;

    public AddMoney(WendingMachine machine,
int money)
    {
        _machine = machine;
        _money = money;
    }
```

```
    public void Execute()
    {
        _machine.AddBalance(_money);
    }
}
```

Заметьте, что прямо над классом указывается наш атрибут. Во-вторых, мы можем получить информацию об атрибутах через рефлексию в момент поиска нужной команды.

Фрагмент 2.43

```
private Type GetCommandTypeByName(string
name)
{
    return GetCommandsTypes()
        .Where(type =>
type.GetCustomAttributes<CommandAttribute>()
                        .Any(attribute =>
attribute.CommandName == name))
        .FirstOrDefault();
}
```

Нам пришлось изменить только метод GetCommandTypeByName. В отличие от старой версии, в Where, мы выбираем только те типы, у которых есть атрибут CommandAttribute со значением CommandName, равным названию искомой команды.

Попутно нам нужно внести правку в метод GetCommands

Фрагмент 2.44

```
public IEnumerable<string> GetCommands()
{
    return GetCommandsTypes()
            .SelectMany(type =>
type.GetCustomAttributes<CommandAttribute>())
            .Select(attribute =>
attribute.CommandName);
}
```

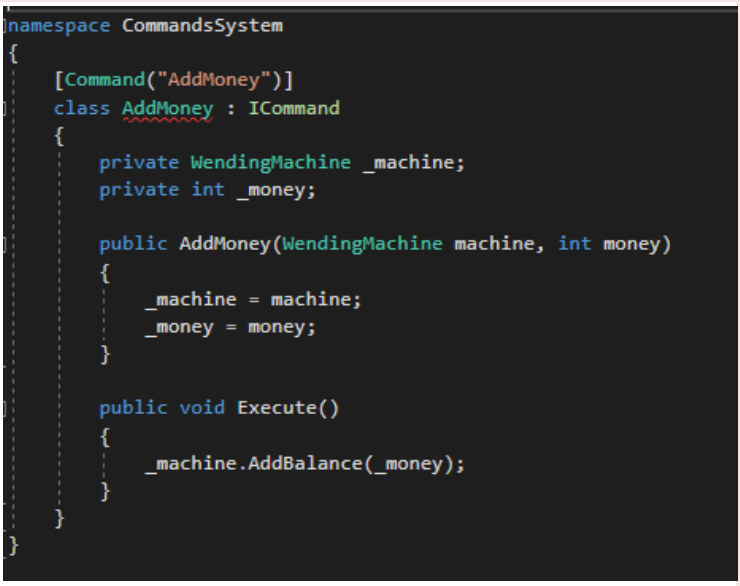
Боремся с дубликатами с помощью Roslyn API

Мы с вами можем получить неявную ситуацию, когда есть два класса помеченных атрибутом Comand с одинаковым названием команды. В таком случае будет выбираться первый найденный класс, что является не совсем очевидным поведением.

Совершенно очевидно, что может быть всего один атрибут с определённым названием. И учитывая это, нам нужно придумать что-то, что поможет избежать ошибок. Один из вариантов – сделать автоматический анализатор кода, который будет проверять наш проект на наличие команд с одним и тем же именем.

Наш анализатор будет строится на основе Roslyn API. Visual Studio умеет запускать пользовательские анализаторы кода, которые могут проводить те или иные проверки.

Анализатор доступен как во время компиляции, так и во время написания кода. Получившаяся у нас утилита подсвечивает классы, которые помечены дублирующимся атрибутом. Делает она это сразу после того, как мы добавляем атрибут. Выглядит это так.

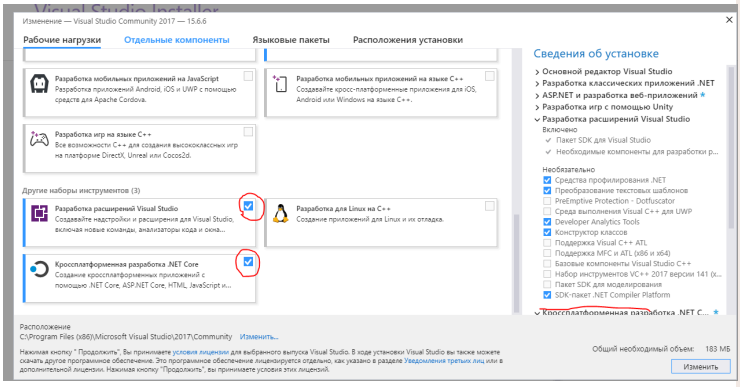


```
namespace CommandsSystem
{
    [Command("AddMoney")]
    class AddMoney : ICommand
    {
        private WendingMachine _machine;
        private int _money;

        public AddMoney(WendingMachine machine, int money)
        {
            _machine = machine;
            _money = money;
        }

        public void Execute()
        {
            _machine.AddBalance(_money);
        }
    }
}
```

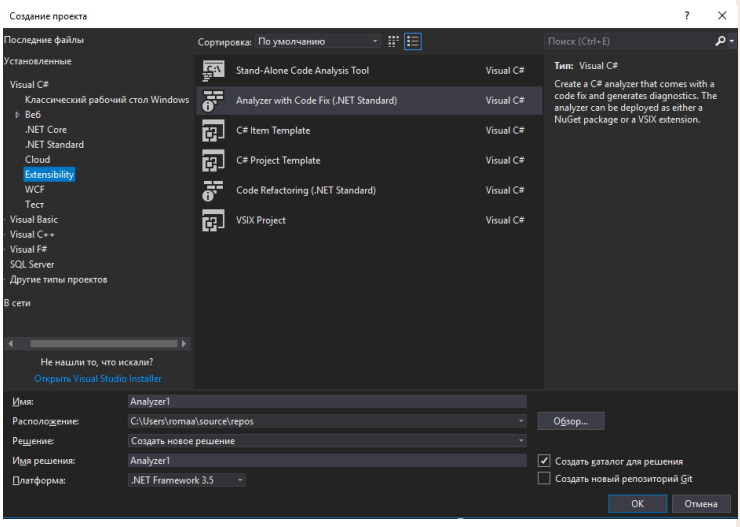
Если навести на ошибку, то мы увидим текст о том, что команда с таким именем уже есть. И это действительно так. Пососедству я сделал класс и пометил его командой с таким же именем.

Начинать разработку своего анализатора нужно с запуска установщика Visual Studio 2017 и доустановки следующих компонентов:

- Разработка расширений VisualStudio:
    - SDK-пакет для .Net Compiler Platform;
    - Кроссплатформенная разработка .Net Core;



После этого, при создании проекта вам будет доступен шаблон “Analyzer With Code Fix”. При его создании, вы получите исчерпывающий шаблон вместе с Unit тестами.



С помощью предоставленного нами API, мы можем создавать проверки, которые могут содержать

исправления кода. Т.е если мы нашли ошибку, то мы можем предложить автоматическое исправление. В нашем случае, я отказался от этого и мы ограничились только сообщением об ошибке.

После того, как анализатор готов, вы можете его скомпилировать и добавить его в разделе “Ссылки” вашего проекта, далее он начнёт работать.

Листинг WendingMachine/Commands/Commands/CommandsAnalyzer.cs

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;

namespace Commands
{

[DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class CommandsAnalyzer :
DiagnosticAnalyzer
    {
```

```
        public const string DiagnosticId =
"CommandDublicate";

        private static readonly
LocalizableString Title = "Command
Dublicate";
        private static readonly
LocalizableString MessageFormat = "Есть
команда с таким же названием";
        private static readonly
LocalizableString Description = "В проекте
есть команда с таким же названием";
        private const string Category =
"Error";

        private static DiagnosticDescriptor
Rule = new DiagnosticDescriptor(DiagnosticId,
Title, MessageFormat, Category,
DiagnosticSeverity.Error, isEnabledByDefault:
true, description: Description);

        public override
ImmutableArray<DiagnosticDescriptor>
SupportedDiagnostics { get { return
ImmutableArray.Create(Rule); } }
        public static readonly string
AttributeName = "CommandAttribute";

        public override void
Initialize(AnalysisContext context)
        {
```

```
context.RegisterSymbolAction(AnalyzeSymbol,
SymbolKind.NamedType);
        }

        private static void
AnalyzeSymbol(SymbolAnalysisContext context)
        {
            var namedTypeSymbol =
(INamedTypeSymbol)context.Symbol;
            var commands = new
GetAllCommandsVisitor(namedTypeSymbol);

commands.Visit(context.Compilation.GlobalName
space);

            var declaredAttributes =
namedTypeSymbol.GetAttributes();
            foreach (var attribute in
declaredAttributes)
            {
                if
(attribute.AttributeClass.Name ==
AttributeName)
                {
                    if
(commands.Commands.Contains(attribute.Constru
ctorArguments[0].ToCSharpString()))
                    {
                        var diagnostic =
Diagnostic.Create(Rule,
```

```
namedTypeSymbol.Locations[0],
namedTypeSymbol.Name);


context.ReportDiagnostic(diagnostic);
                    }
                }
            }
        }
    }

    public class GetAllCommandsVisitor :
SymbolVisitor
    {
        public List<string> Commands = new
List<string>();
        public INamedTypeSymbol Ignored;

        public
GetAllCommandsVisitor(INamedTypeSymbol
ignored)
        {
            Ignored = ignored;
        }

        public override void
VisitNamespace(INamespaceSymbol symbol)
        {

Parallel.ForEach(symbol.GetMembers(), s =>
s.Accept(this));
```

```
        }

        public override void
VisitNamedType(INamedTypeSymbol symbol)
        {
            if (symbol != Ignored)
            {
                foreach (var attribute in
symbol.GetAttributes())
                {
                    if
(attribute.AttributeClass.Name ==
CommandsAnalyzer.AttributeName)
                    {

Commands.Add(attribute.ConstructorArguments[0
].ToCSharpString());
                    }
                }
            }
        }
    }
}
```

На первый взгляд, листинг довольно сложный. Но на самом деле это конечно же не так. Мы просто имеем некий код, который выполняется для каждого класса (фактически для каждого именованного типа) и некоторую обвязку, для его выполнения.

В корне у нас лежит этот метод.

Фрагмент 2.45

```
public override void
Initialize(AnalysisContext context)
{

context.RegisterSymbolAction(AnalyzeSymbol,
SymbolKind.NamedType);
}
```

Он сообщает ядру о том, что на каждый тип, определенный в проекте, к которому подключен этот анализатор, нужно выполнить специальный код, который может предоставить диагностическую информацию, а в нашем случае ошибку.

С помощью SymbolKind мы можем указать какого вида сущности мы хотим анализировать.

Фрагмент 2.46

```
private static void
AnalyzeSymbol(SymbolAnalysisContext context)
{
    var namedTypeSymbol =
(INamedTypeSymbol)context.Symbol;
    var commands = new
GetAllCommandsVisitor(namedTypeSymbol);

commands.Visit(context.Compilation.GlobalName
space);
```

```
    var declaredAttributes =
namedTypeSymbol.GetAttributes();
    foreach (var attribute in
declaredAttributes)
    {
        if (attribute.AttributeClass.Name ==
AttributeName)
        {
            if
(commands.Commands.Contains(attribute.Constru
ctorArguments[0].ToCSharpString()))
            {
                var diagnostic =
Diagnostic.Create(Rule,
namedTypeSymbol.Locations[0],
namedTypeSymbol.Name);


context.ReportDiagnostic(diagnostic);
            }
        }
    }
}
```

Логика очень проста, с помощью посетителя мы просматриваем рекурсивно всю сборку и ищем типы, у которых есть атрибут с названием “Command” и собираем все названия команд, игнорируя команду типа, который сейчас анализируем.

Далее, просто проверяем, что если на анализируемом типе висит атрибут с названием команды, которая уже есть в другому месте, мы составляем репорт о том, что ошибка в такой-то точке с такими-то подробностями.

Код нашего визитера и вообще схема взаимодействия с деревом сущностей может оказаться вам интересной. Это хороший пример использования паттерна Visitor.

Фрагмент 2.47

```
public class GetAllCommandsVisitor :
SymbolVisitor
{
    public List<string> Commands = new
List<string>();
    public INamedTypeSymbol Ignored;

    public
GetAllCommandsVisitor(INamedTypeSymbol
ignored)
    {
        Ignored = ignored;
    }

    public override void
VisitNamespace(INamespaceSymbol symbol)
    {
```

```
        Parallel.ForEach(symbol.GetMembers(),
s => s.Accept(this));
    }

    public override void
VisitNamedType(INamedTypeSymbol symbol)
    {
        if (symbol != Ignored)
        {
            foreach (var attribute in
symbol.GetAttributes())
            {
                if
(attribute.AttributeClass.Name ==
CommandsAnalyzer.AttributeName)
                {

Commands.Add(attribute.ConstructorArguments[0
].ToCSharpString());
                }
            }
        }
    }
}
```

Roslyn API очень богат и я не берусь описывать его в этом томе. Я надеюсь, что описанное здесь, вас заинтересует и раскроет идею о том, что нужно пытаться бороться с неявными местами и делать всё возможное, чтобы помочь избежать ошибки тем, кто будет работать после вас.

Выдача сдачи. Часть 1.

Ваша задача

Одна из важнейших задач в рамках разработки вендингового аппарата – это выдача сдачи наименьшим количеством монет. Сейчас выдача сдачи реализована заглушкой, которая полностью обнуляет баланс.

Мы можем с вами притянуть фактическую необходимость в выдаче сдачи монетами. Но на самом деле, эта задача нам интересна с точки зрения темы динамического программирования, которую мы попробуем рассмотреть в рамках такой интересной задачи.

Давайте четко сформулируем задачу, чтобы понять что делать. Задачу я разделю на две части, первая будет заключаться в более теоретическом поле и не будет напрямую решать задачу выдачи конкретных монет. Вторая часть, на наработках первой, будет реализовывать конкретное поведение.

Сейчас вам нужно сделать следующее:

- У вас есть набор номиналов монет (1, 5, 7, 10, 15);

- У вас есть сдача, которую нужно выдать (120);
- Вам нужно найти минимальное количество монет для выдачи сдачи;

Подсчет чисел Фибоначчи

Начнём мы, на первый взгляд, с отвлеченной темы. Как на счёт того, чтобы написать на C# программу, которая выдает нам число N в последовательности Фибоначчи?
Эта последовательность обычно определяется так, что первые два числа либо 1 и 1, либо 0 и 1. А все последующие – это сумма двух предыдущих чисел.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, ...

Алгоритм нахождения мы запакуем в такую функцию:

Фрагмент 2.48

```
static int Fibonacci(int n)
{
        ...
}
```

Мы ей передаем N, а он нам число из последовательности под этим номером. Пример:

- 0 = 0;
- 1 = 1;
- 2 = 1;
- 3 = 2;
- 4 = 3;

- 5 = 5;
- 6 = 8;

Эту последовательность мы можем определить через рекуррентное отношение.

Выглядит оно вот так:

$$F_n = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ F_{n-1} + F_{n-2}, & n \geqslant 2. \end{cases}$$

Реализация формулы через рекурсию

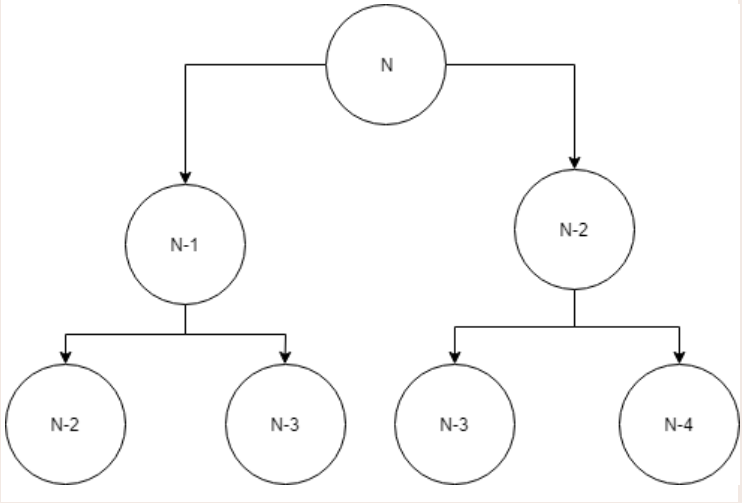
Мы можем сделать реализацию в лоб на основе рекурсии. Формула нам говорит, что любое входящее число N – это сумма чисел N-1 и N-2. И есть базовые случаи при N равным 0 и 1. Мы упростим базовый случай сказав, что при любом N, меньше 3, результат будет равен 1.

В результате мы получим такую реализацию.

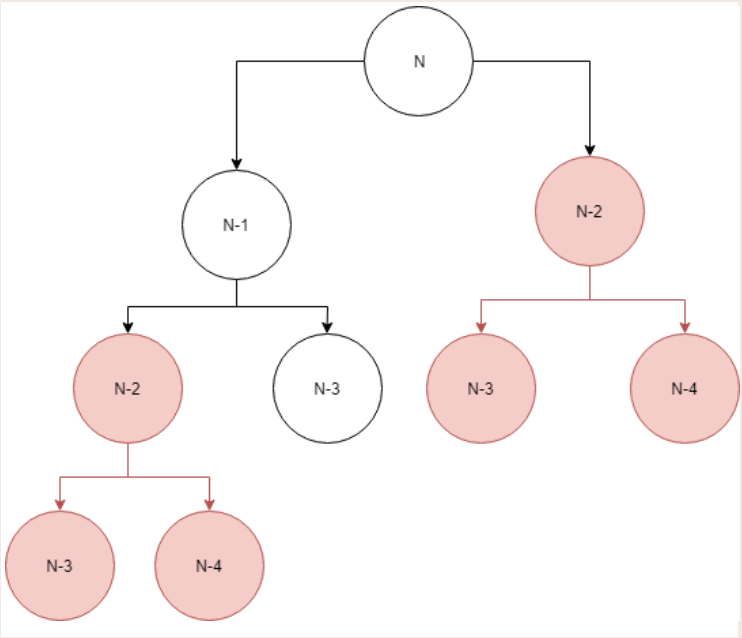
Фрагмент 2.49

```
static int Fibonacci(int n)
{
    if (n < 3) return 1;
    return Fibonacci(n - 1) + Fibonacci(n -
2);
}
```

Она дает правильный ответ, но не за адекватное время. К сожалению, рост количества вызовов функций – экспоненциальный. Чтобы понять это, можно посмотреть на граф вызовов.
Функция работает таким образом, что на вызов от N происходит множество подобных вызовов внутри себя (рекурсивно), но с другими N. В процессе рекурсии мы очень часто повторяемся и множество раз рассчитываем N повторно.



На этом графе видно, что ветви рекурсии содержат идентичные вызовы, это и не позволяет получить приемлемое время выполнения при большом N. Чем сильней разрастается граф, тем нам понятней, что его можно очень сильно оптимизировать.


N
N-1
N-2
N-2
N-3
N-3
N-4
N-3
N-4

Динамическое программирование

Ситуация описанная выше не такая уж и редкая, и её можно встретить во многих задачах. В том числе и задачи про монетки. И чтобы быть готовыми написать алгоритм сдачи, которые работает быстро и четко, нам нужно разобраться с проблемами чисел Фибоначчи

Динамическое программирование – это метод оптимизации алгоритмов, в основе которого лежат рекуррентные уравнения, по типу того что я приводил выше при описание последовательности Фибоначчи. По сути слово программирование, здесь не имеет в общеупотребительном толкование смысла. Тут уместней будет заменить его на “оптимизация”.

Для использования этой методики нам нужно разбить задачу на подзадачи до того момента пока мы не дойдём до примитивного решения которое можно решить здесь и сейчас. А потом объединить решения и тем самым решить основную задачу.

Это налагает определенные ограничения. Мы можем использовать этот метод только если есть:

1. Оптимальная подструктура.
2. Перекрывающиеся подзадачи.

Оптимальная подструктура

Это означает что если мы правильно решим подзадачи, то сможем на основе этого правильно решить основную задачу. Если наша основная задача найти F(N), где F функция которая возвращает N-ое число в последовательности. И мы определили эту функция как F(N) = F(N-1) + F(N-2) то тогда мы можем говорить об оптимальной подструктуре.

Если мы найдём решение для F(N-1) и F(N-2), то на основе этого мы сможем построить оптимальное решение. Оптимальная подструктура может отсутствовать, из-за этого мы не можем применять данную методику.

Например, если перед нами стоит задача

Перекрывающиеся подзадачи

При решение основной задачи мы встречаемся с повторяющимся подзадачами. Если посмотреть на наш граф выполнения, приведённый выше. Мы заметим что во-время решения основной задачи F(N) мы часто прибегаем к повторном решению подзадач.

Мы можем обнаружить что при F(N) мы два раза будем искать F(N-2) что в свою очередь приведёт к повторному поиску F(N-3) и F(N-4). Техники динамической оптимизации нацелены на

исправление этого недостатка с помощью двух способов – нисходящего и восходящего решения.

Два пути

Нисходящий – мемоизация

Первый путь очень прост, на самом деле всё описываемое здесь просто, хоть и имеет серьезную научную основу. Нисходящий подход заключается всё в том же рекурсивном описании, но здесь неспроста стоит слово “мемоизация”.

Смысл заключается в том, что мы запоминаем предыдущее решение, и если мы уже решали задачу F(N-2) то повторно её решать не будем и возьмем решение которые мы запомнили. Это нас избавит ещё от повторного решение F(N-3) и F(N-4) и т.д.

Фрагмент 2.50

```
private static Dictionary<int, int>
_fibonacciResults = new Dictionary<int,
int>();

static int FibonacciMemoization(int n)
{
    if(_fibonacciResults.TryGetValue(n, out
int value))
    {
        return value;
```

```
    }
    else
    {
        if (n < 3) _fibonacciResults.Add(n,
1);
        else
        {
            _fibonacciResults.Add(n,
FibonacciMemoization(n - 1) +
FibonacciMemoization(n - 2));
        }
    }

    return _fibonacciResults[n];
}
```

В этой реализации мы использовали словарь входящий в пространство имен System.Collections.Generic. С его помощью мы запоминаем предыдущие решения, и возвращаем их если они у нас имеются. Такое решение работает в разы быстрей хоть и занимает больше памяти. Это нормально, при оптимизации есть золотой закон – улучшая скорость работы мы начинаем тратить больше памяти.

В подобных случаях не всегда есть смысл использовать именно словарь. Здесь он приведён

только для того, чтобы более наглядно показать, что мы “запоминаем” некоторое решение.

При использования подхода сверху вниз, порядок решения задач опять-таки диктуется сверху. Этот способ на самый оптимальный, но зачастую подходящий. По этому на нём всегда можно остановится если он выдал приемлемый результат.

Восходящий – табуляция

В предыдущий раз мы шли сверху вниз и спускали до примитивного решения. Сейчас пойдём от обратного, и пойдем снизу вверх воспользовавшись техниками восходящего анализа.

Мы заранее решаем подзадачи и так же откладываем их решение. Этот подход быстрей за счёт того, что нет накладных расходов на вызов функций. Но в некоторых случаях мы можем решить больше задач чем на требовалось бы для поиска оптимального решения.

Суть заключается в том, что мы сначала находим решение для F(0), F(1) потом F(2) и т.д пока не дойдём до F(N). Когда мы окажемся в F(N) мы будем точно знать что у нас есть решения F(N-1) и F(N-2)

Фрагмент 2.51

```
static int Fibonacci(int n)
{
    int[] data = new int[n + 1];
    data[0] = 0;
    data[1] = 1;

    for(int i = 2; i < data.Length; i++)
    {
        data[i] = data[i - 1] + data[i - 2];
    }

    return data[n];
}
```

Сходное решение именно этой задачи, можно представить ещё таким образом.

Фрагмент 2.52

```
static int Fibonacci(int n)
{
    if (n < 3) return 1;

    int result = 1;
    int old = 1;

    for(int i = 2; i < n; i++)
    {
        int temp = old;
        old = result;
```

```
            result += temp;
        }

        return result;
    }
```

Оно отличается от предыдущего тем, что мы не запоминаем все решения, а только те, что нам понадобятся далее. И данная реализация не универсальна и возможна только в нашей ситуации. Подобные решения могут получатся сами по себе, без осмысления задачи в терминах динамического программирования и в этом нет ничего страшного.

Возможное решение. Алгоритм выдачи сдачи с оптимизацией на C#.

Решение этой задачи зависит от некоторых входных условий. Если в нашем распоряжении монеты определенного номинала, то мы можем свести решение к последовательному делению суммы сдачи на все возможные номиналы, начиная с самого большого.

Если вы не забыли, задача состоит в том, чтобы найти минимальное количество монет, чтобы выдать сдачу. Оформлять варианты будем в такую или похожую функцию. Первым аргументом присылаем массив номиналов, а вторым сумму сдачи.

Фрагмент 2.53

```
static int GetChange(int[] values, int
change)
{
...
}
```

Решение на основе деления

Мы можем отсортировать номиналы от большего к меньшему и начать делить сумму сдачи на все значения по очереди, а результат деления каждый

раз приплюсовывать к какой-то переменной. По сути, результат деления суммы сдачи на номинал указывает на то, сколько монет мы сможем выдать. Но также может возникать остаток от деления, его мы присваиваем в сумму сдачи. И делим на следующий номинал. Так, постепенно, мы придем к какому-нибудь ответу.

Полностью решение представлено ниже.

Фрагмент 2.54

```
static int GetChangeSimple(int[] values, int
change)
{
    int count = 0;
    foreach(int value in
values.Distinct().OrderByDescending(x => x))
    {
        count += change / value;
        change = change % value;
        if (change == 0) return count;
    }

    return 0;
}
```

Как вы видите оно очень простое, а благодаря LINQ его удалось записать в 6 строчек. Но решает ли оно нашу задачу? Да, если номиналы подчиняются

определённому правилу. Так, в случае если у нас номиналы {1, 5, 10, 15, 20} и сумма сдачи скажем 50, то всё правильно, он нам сообщит о том, что нужно 3 монеты: 20+20+10.

Но что будет, если номиналы мы зададим как {1, 5, 10, 15, 17, 20}, а сумму сдачи как 34. Сюрприз-сюрприз, алгоритм насчитал 6 монет: 20 + 10 + 1 + 1 + 1 + 1. Так ли это? Нет, сдачу можно выдать двумя монетами: 17 + 17. В чём проблема алгоритма? В том, что он работает жадным методом.

При решение задачи, в таких условиях, выдача сдачи монетами номиналом в 20 может выглядеть хорошо, но это отрежет нам более оптимальный путь решения. В результате, этот алгоритм не самый универсальный и мы попробуем придумать кое-что по-лучше.

Рекурсивное решение

Более совершенный алгоритм будет строиться, по-сути, на основе перебора. Мы будем анализировать разные пути размена до тех пор, пока не найдём оптимальный.

Рекуррентное соотношение можно записать так:

$$f(n) = \begin{cases} 1, & n \subset S; \\ min\left\{1 + f(n-s), s \subset S;\right\}, & n \notin S. \end{cases}$$

Мы определили функцию, которая работает с неким множеством S, которое в нашем случае является набором доступных номиналов. На входе эта функция получает информацию о том, сдачу для какой суммы нужно дать, а её результатом является количество монет.

Если сумма сдачи может быть дана одной монетой, то возвращаем единицу. Это базовый случай и он не требует дальнейшего вычисления. Делается это через проверку принадлежности n множеству S (сумма сдачи равняется одному из номиналов монет).

Если мы не можем дать сдачу одной монетой, то мы начинаем впадать в рекурсию. Мы берём каждый номинал и отнимаем его от текущей сдачи и вызываем функцию с получившимся значением. А к результату прибавляем единицу. В результате у нас будет множество количества монет для сдачи, если мы начнем выдачу с определенного номинала и нам нужно взять минимальное значение.

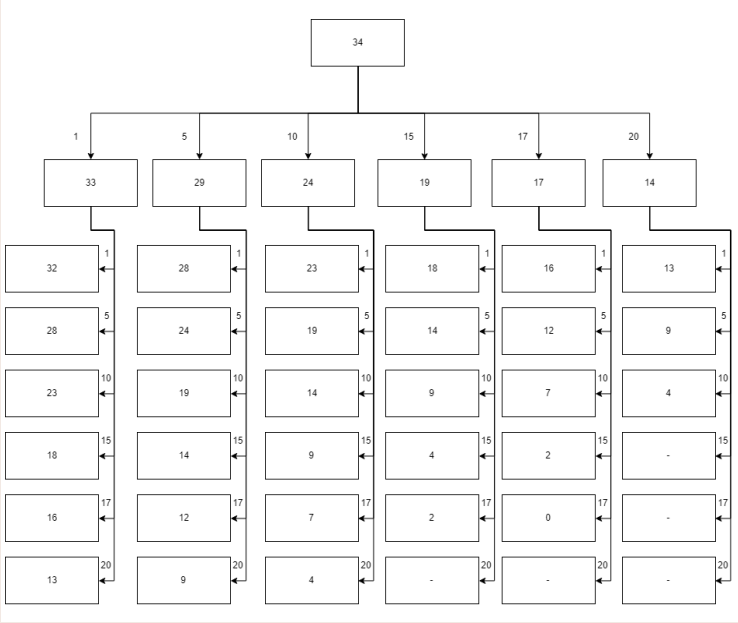
Предположим что S = { 1, 5, 10, 15, 17, 20 }.

И мы вызовем функцию со значением 34.

В таком случае, мы получим в первом шаге вызовы следующих функций:

F(33), F(29), F(24), F(19), F(17), F(14). Каждый из этих вызовов приведёт ещё к 6 вызовам и т.д. В результате получится огромный граф вариантов сдачи. Кратчайший путь от самого верха до базового случая (точки, где мы получим ноль) и есть ответ.

Этот путь вычисляет по ходу дела.



Давайте посмотрим на реализацию.

Фрагмент 2.55

```
static int GetChange(int[] values, int
change)
{
```

```
    int minCoins = change;

    if (values.Contains(minCoins))
    {
        return 1;
    }
    else
    {
        foreach (var value in values.Where(x
=> x <= change))
        {
            int numCoins = 1 +
GetChange(values, change - value);
            if(numCoins < minCoins)
            {
                minCoins = numCoins;
            }
        }

        return minCoins;
    }
}
```

Данный алгоритм более гибкий и надёжный. Но, как вы уже поняли, он не адекватен с точки зрения ресурсов. Очень быстро граф разрастается до немыслимых размеров и мы уже не можем быстро найти правильный ответ. И, как вы уже догадались, мы попробуем применять динамическое программирование для решения задачи.

Рекурсивное решение – оптимизация на основе мемоизации

Если ещё раз посмотреть на граф выполнения выше, мы обнаружим что очень часто мы прибегаем к повторному расчёт сдачи одних и тех же сум. Это отрицательно сказывается на скорости выполнения, и мы можем прямо сейчас применить технику мемоизации и просто запомнить решения подзадач.

Давай рассмотрим реализацию.

Фрагмент 2.56

```
static int GetChangeMemoization(int[] values,
int change, int[] getChangeResults)
{
    int minCoins = change;

    if (values.Contains(minCoins))
    {
        getChangeResults[change] = 1;
        return 1;
    }
    else if (getChangeResults[change] != 0)
    {
        return getChangeResults[change];
    }
    else
```

```
    {
        foreach (var value in values.Where(x
=> x <= change))
        {
            int numCoins = 1 +
GetChangeMemoization(values, change - value);
            if (numCoins < minCoins)
            {
                minCoins = numCoins;
                getChangeResults[change] =
minCoins;
            }
        }
    }
    return minCoins;
}
```

Мы не сделали ничего выдающегося, просто запомнили предыдущие решения. Но алгоритм уже стал гораздо более резвым, и при увеличение числа сдачи он уже не уходит в очень долгие раздумья.

Рекурсивное решение – оптимизация на основе табуляции

Как было описано раньше, этот подход оптимизации заключается в том, что мы заранее решаем подзадачи. Мы заполняем таблицу результатов с самого начала до нужной суммы.

Каждый раз мы работаем с некой суммой N и пытаемся найти решение для неё. Для этой N мы смотрим: если взять некое предыдущее решение для N минус номинал доступной монеты, то не будет ли решение, полученное на основе него, (к тому решению мы добавим 1, что означает добавление монеты этого номинала) более оптимальным? Если да, то для N мы выбираем именно его.

Давайте посмотрим пошаговую работу алгоритма. Зададим номиналы как { 1, 5, 7, 10}. А сумму сдачи, равную 14. При запуске список результатов будет такой.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Сверху – сумма сдачи, снизу – количество монет.
Далее рассмотрим пошагово как алгоритм заполнит таблицу.
1) Сдачу, суммой в 1, мы можем дать одной монетой, номиналом в 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14

1	0	0	0	0	0	0	0	0	0	0	0	0	0

2) Сдачу, суммой в 2, мы можем дать двумя монетами, номиналом в 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14

1	2	0	0	0	0	0	0	0	0	0	0	0	0

3) Сдачу, суммой в 3, мы можем дать тремя монетами, номиналом в 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14

1	2	3	0	0	0	0	0	0	0	0	0	0	0

4) Сдачу, суммой в 4, мы можем дать четырьмя монетами, номиналом в 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	0	0	0	0	0	0	0	0	0	0

5) И вот тут начинается самое интересное. Находясь в сумме 5, мы имеем в распоряжении уже две монеты, номиналом в 1 и 5. Мы проверяем два решения: посмотрев на решение суммы 5 – 1 монеты, получаем 4 и на сумму 5-5, получаем ноль. Ноль меньше четырёх, поэтому мы используем его + 1 монета. И получаем решение в 1 монету.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	0	0	0	0	0	0	0	0	0

6) Здесь мы проделываем тот же трюк, только приходим к тому, что решение для суммы, меньше нашей на монету, номиналом 1, нам подходит больше.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	0	0	0	0	0	0	0	0

7)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	0	0	0	0	0	0	0

8)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	0	0	0	0	0	0

9)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	0	0	0	0	0

10)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	1	0	0	0	0

11)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	1	2	0	0	0

12)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	1	2	3	0	0

13)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	1	2	3	4	0

14)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	1	2	1	2	3	1	2	3	4	1

И полная реализация алгоритма.

Фрагмент 2.57

```
static int GetChangeTabulization(int[]
values, int change, int[] getChangeResults)
{
    for(int cents = 0; cents < change+1;
cents++)
    {
        int coinsCounts = cents;
        foreach (var value in
values.Where(value => value <= cents))
        {
            if(getChangeResults[cents-value]
+ 1 < coinsCounts)
            {
                coinsCounts =
getChangeResults[cents - value] + 1;
            }
        }
        getChangeResults[cents] =
coinsCounts;
    }

    return getChangeResults[change];
}
```

Завершение

Вот и подходит к концу первый том. Ставя точку я открываю новый черновик чтобы продолжить.